University of Zurich[UZH]

Department of Informatics

Continuous Experimentation for Software Developers

Dissertation submitted to the Faculty of Business,
Economics and Informatics
of the University of Zurich

to obtain the degree of
Doktor der Wissenschaften, Dr. sc.
(corresponds to Doctor of Science, PhD)

presented by
Gerald Schermann
from Austria

approved in July 2019

at the request of
Prof. Dr. Harald C. Gall
Dr. Philipp Leitner
Prof. Dr. Elisabetta Di Nitto

University of Zurich[UZH]

The Faculty of Business, Economics and Informatics of the University of Zurich hereby authorizes the printing of this dissertation, without indicating an opinion of the views expressed in the work.

Zurich, July 17, 2019

Chairman of the Doctoral Board: Prof. Dr. Thomas Fritz

Acknowledgments

First and foremost, I thank my advisors Philipp Leitner and Harald Gall. Not only their continuous guidance and support but also their expertise was invaluable during the course of my doctoral studies. Thank you for giving me the opportunity to pursue a PhD paired with the freedom to work on topics that interest me the most. I thank Elisabetta Di Nitto for serving on my PhD committee as an external examiner, dedicating her valuable time for evaluating my work, and the great feedback that I received.

Special thanks go to Erik Wittern and Fábio Oliveira who gave me the opportunity for doing an internship at IBM Research in New York. Thanks for the great discussions and valuable feedback.

I especially thank my current and former colleagues from the Software Evolution and Architecture Lab (SEAL), without them this adventure would have been only half the fun: Carol Alexandru, Martin Brandtner, Adelina Ciurumelea, Thomas Fritz, Christian Inzinger, André Meyer, Sebastiano Panichella, Sebastian Proksch, Manuela Züger, and in particular Jürgen Cito, Giovanni Grano, Katja Kevic, Christoph Laaber, Sebastian Müller, and Carmine Vassallo. Being it endless discussions during coffee breaks or lunches, playing Uno and Exploding Kittens, or all the other fun we had in our offices, at retreats, or on conference trips, all these little fragments made the time of my PhD memorable.

Finally, I want to thank my family for their unconditional support and encouragement during my studies.

Gerald Schermann
Zürich, July 2019

Abstract

For staying competitive in highly-contested and fast-growing markets such as the Web companies need to continuously adapt their software. Releasing incremental changes fast, while at the same time guaranteeing high quality, requires release processes that are strongly based on tools to automate software build, test, and deployment. While previous methods of releasing changes hardly involved evidence to support decisions (e.g., do users appreciate my new feature?), nowadays, sophisticated telemetry solutions keep track of releases and captured live production data has become the basis for data-driven decision making. High automation bundled with telemetry promotes the advent of *continuous experimentation* practices (e.g., canary releases, or A/B testing) that guide development activities based on data collected on a fraction of the user population on a new *experimental* version of the software in the production environment. However, adopting continuous experimentation to move towards data-driven decision making is not a straightforward process, it involves setting up a complex experimentation infrastructure and requires methods and tools to cover the entire life cycle of experiments, from their design to the assessment of their outcome.

In the context of this thesis, we address challenges surfacing within experiment life cycle phases with the goal to devise research approaches to support the thesis statement: *"A detailed understanding of the characteristics of continuous experiments enables building a conceptual framework for planning, executing, and analyzing experiments"*. To pay attention to the trend towards decentralized microservice teams independently running experiments, our approaches are tailored to software developers and release engineers and designed to foster the parallel execution of experiments with as little overhead as possible, to identify

optimal plans to collect required sample sizes for sound statistical interpretation, and to provide means for experiment health assessment.

Informed by the findings from an empirical study on the state of practice, we characterized experimentation practices into *regression-driven* experiments (e.g., canary releases) and *business-driven* experiments (e.g., A/B testing), and derived a conceptual framework for experimentation. This framework built the basis for three research approaches and prototypes as concrete instantiations that have been extensively validated through numerical experimentation:

Fenrir targets the planning phase of experiments and scheduling in particular. We formulate scheduling as an optimization problem with the aim of fostering the parallel execution of experiments, while at the same time ensuring that enough data is collected for every experiment and the collected data is not skewed by overlapping experiments. Fenrir outperforms other approaches not only in the quality of the schedules identified but also in terms of execution time.

Bifrost supports the execution phase and involves the automated, data-driven execution of multi-phased experiments (e.g., an A/B test follows a canary release). Experiments are specified in a domain-specific language and our concept of conditional chaining allows triggering automated actions such as rollbacks in case of spotted irregularities. Bifrost supports running more than a hundred experiments in parallel without introducing a significant performance degradation.

Finally, we investigated approaches and devised a research prototype for experiment health assessment with the goal of raising the developer's awareness about (topological) changes in the context of experiments. We characterized change types that surface within the evolution of microservice-based applications and developed and evaluated multiple heuristics to rank identified changes according to their potential impact on the application's health state.

Overall, we demonstrated that our framework enables planning, executing, and analyzing large-scale continuous experiments. There are multiple opportunities for future work to extend our framework and approaches including smarter experimentation platforms that dynamically decide how experimentation logic is executed, visualization extensions to IDEs (integrated developer environments), and providing means for experiment verification based on statistical models.

Zusammenfassung

Um Wettbewerbsfähigkeit in hart umkämpften und schnell wachsenden Märkten wie dem Web zu gewährleisten, ist es für Unternehmen essentiell, fortlaufend ihre Software anzupassen. Schnelle und kontinuierliche Releases von inkrementellen Änderungen bei gleichzeitiger Gewährleistung einer hohen Qualität, erfordern Releaseprozesse, die stark auf Werkzeuge zur Automatisierung von Software Builds, Tests und Deployments basieren. Während bisherige Releaseverfahren kaum Echtzeitinformationen zur Entscheidungsfindung herangezogen haben (beispielsweise, schätzen Benutzer mein neues Feature?), überwachen heutzutage ausgefeilte Telemetrielösungen Software Releases und die dabei gesammelten Echtzeitinformationen bilden die Basis für datengetriebene Entscheidungsfindung. Telemetrie gebündelt mit einem hohen Grad an Automatisierung fördert das Aufkommen von *Continuous Experimentation* Praktiken (z.B. Canary Releases oder A/B-Tests), welche Entwicklungstätigkeiten auf der Grundlage von Informationen lenken, die für einen Bruchteil der Benutzerpopulation an einer neuen *experimentellen* Version der Software direkt in der Produktivumgebung gesammelt werden. Die Einführung von Continuous Experimentation mit dem Ziel der datengetriebenen Entscheidungsfindung ist jedoch kein simpler Prozess. Die Einführung erfordert nicht nur den Aufbau einer komplexen Infrastruktur, sondern auch Methoden und Werkzeuge, die den gesamten Lebenszyklus von Continuous Experiments abdecken, von der Planung bis zur Ergebnisevaluation.

Im Rahmen dieser Dissertation beschäftigen wir uns mit Herausforderungen, welche in den verschiedenen Phasen der Lebenszyklen von Experimenten auftreten, mit dem Ziel, Forschungsansätze zu entwickeln, die die These dieser

Dissertation unterstützen: *"Ein umfassendes Verständnis der Charakteristiken von Continuous Experiments ermöglicht die Umsetzung eines konzeptuellen Frameworks für das Planen, Durchführen und Analysieren von Experimenten"*. Um dem Trend zu dezentralisierten Microservice-Teams Rechnung zu tragen, die unabhängig voneinander Experimente durchführen, sind unsere Ansätze auf Softwareentwickler und Release Engineers zugeschnitten. Darüber hinaus sollen diese Forschungsansätze die parallele Durchführung von Experimenten mit möglichst geringen Leistungseinbussen unterstützen, Wege identifizieren um erforderliche Stichprobengrössen für robuste statistische Auswertung zu gewährleisten und Möglichkeiten für eine aussagekräftige Beurteilung des Zustandes von Experimenten bieten.

Die Erkenntnisse, welche in einer empirischen Studie über den Stand der Praxis zu Continuous Experimentation gewonnen wurden, haben zur Charakterisierung von Praktiken in *regressionsgetriebenen Experimenten* (z.B. Canary Releases) und *businessgetriebenen Experimenten* (z.B. A/B Tests) geführt, als auch zur Entwicklung eines konzeptuellen Frameworks für Continuous Experimentation. Dieses Framework bildet die Grundlage für drei Forschungsansätze und Prototypen als konkrete Umsetzungen, die durch numerische Methoden umfassend validiert wurden:

Fenrir ist auf die Planungsphase von Experimenten und konkret auf Scheduling ausgelegt. Wir formulieren Scheduling als Optimierungsproblem mit dem Ziel, die parallele Ausführung von Experimenten zu fördern, während wir gleichzeitig sicherstellen, dass für jedes Experiment genügend Daten gesammelt werden und die gesammelten Daten nicht durch überlappende Experimente verfälscht werden. Fenrir übertrifft andere Ansätze nicht nur hinsichtlich der Qualität der generierten Schedules, sondern auch hinsichtlich der Ausführungszeit.

Bifrost unterstützt die Ausführungsphase im Lebenszyklus von Experimenten und beinhaltet die automatisierte, datengetriebene Durchführung von mehrphasigen Experimenten, beispielsweise wenn ein A/B-Test einem Canary Release folgt. Die Experimente werden in einer domänenspezifischen Sprache beschrieben und unser Konzept der bedingten Verkettung (*conditional chaining*) unterstützt das Auslösen automatisierter Aktionen, wie beispielsweise

Rollbacks, wenn Unregelmäßigkeiten erkannt werden. BIFROST ermöglicht die parallele Durchführung von mehr als hundert Experimenten ohne erkennbarer Leistungsbeeinträchtigung.

Schliesslich untersuchten wir Ansätze und entwickelten einen Forschungsprototypen für die Bewertung des Zustands von Experimenten, mit dem Ziel, das Bewusstsein von Entwicklern für (topologische) Veränderungen im Kontext von Experimenten zu schärfen. Wir haben verschiedene Arten von topologischen Änderungen charakterisiert, die im Laufe der Evolution von Microservice-basierten Anwendungen zum Vorschein kommen. Darüber hinaus haben wir Heuristiken entwickelt und evaluiert, um solche identifizierten topologischen Änderungen anhand ihres potentiellen Einflusses auf den Zustand von Experimenten und der Gesamtanwendung einzuordnen.

Insgesamt haben wir demonstriert, dass unser Framework die Planung, Durchführung und Analyse von komplexen Continuous Experiments ermöglicht. Wir haben mehrere Möglichkeiten für weiterführende Forschung aufgezeigt, um unser Framework und unsere Ansätze zu erweitern. Dies beinhaltet die Entwicklung intelligenter Plattformen für die Durchführung von Experimenten, welche dynamisch entscheiden, wie die Experimentlogik integriert und ausgeführt wird, Erweiterungsmöglichkeiten hinsichtlich der Visualisierung in Entwicklungsumgebungen und Möglichkeiten zur Verifizierung von Experimenten auf Basis statistischer Modelle.

Contents

List of Figures

List of Tables

1

Synopsis

Continuously adapting software, being it rapidly delivering code changes to fix problems or satisfy newly surfaced requirements, is key for companies to survive in highly-competitive, fast-growing markets such as the Web. The ever existing need to stay ahead of competition [Chen, 2015] bundled with demand for faster innovation (e.g., reduced time-to-market) [Parnin et al., 2017] is fueling the adoption of DevOps practices [Bass et al., 2015] in many companies. DevOps promotes the *continuous deployment* model [Savor et al., 2016] intended to shorten the time between the commit of a code change to a source code repository and the code becoming actively executed in the company's production environment. Successful DevOps installations involve tearing down traditional barriers between development, quality assurance, and operations teams, escaping long established responsibility silos and its accompanied "throwing code over the wall" mentalities, and introducing a new set of software-development methodologies strongly based on tools to automate software build, test, configuration, and deployment processes. This underlying automation drives and enables industry leaders such as Facebook [Feitelson et al., 2013], Microsoft [Kevic et al., 2017], Google [Tang et al., 2010], or Netflix [Gomez-Uribe and Hunt, 2016] to build software ecosystems characterized by hundreds or even thousands of deployments and releases a day. The advent of release process tooling transforms product development to an experimental approach [Humble et al., 2015], or as coined by Parnin et al. [2017] "every feature is an experiment". While previous methods of releasing new features hardly involved evidence to support decisions (e.g., do my users like my new feature?), nowadays, sophisticated monitoring and telemetry solutions keep

track of releases and the captured live production data has become the basis for data-driven decision making. These so called *continuous experimentation* practices (e.g., canary releases [Humble and Farley, 2010], A/B testing [Kohavi et al., 2013], or dark launches [Feitelson et al., 2013]) guide development activities based on data collected on a subset of the user population on a new *experimental* version of the software in the production environment. If the software's performance and correctness remain acceptable, the new version is gradually exposed to more users. However, if it fails to perform as expected, users are shifted back to the previous, stable version to keep the impact of malfunctioning releases low. In contrast to traditional software testing disciplines (e.g., unit testing [Beck, 1999], load testing [Menascé, 2002]), continuous experimentation does not mimic user behavior, it rather uses real users' interactions with the system and the thereby accrued traffic to guide development decisions or reveal performance problems of newly deployed functionality. Thus, continuous experimentation pairs fast insights from live data (i.e., taking advantage of early customer feedback [Chen, 2015]) with manageable risks, and if things go wrong "just" a fraction of the users is affected. A detailed overview of continuous experimentation practices is provided in Section 2.2.1.

Striving for automation is not a straightforward task as recent research revealed. Companies not only face technical and organizational challenges (e.g., Leppanen et al. [2015] and Olsson et al. [2012]), also changed team structures and roles, autonomy, and release pressure impose social challenges [Claps et al., 2015]. Going a step further, adopting continuous experimentation to move towards data-driven decision making involves dealing with a complex experimentation infrastructure (e.g., Xu et al. [2015] and Fabijan et al. [2017]) and requires approaches and tools to cover the entire experiment life cycle from design to analysis. Based on a motivating example, we will showcase key challenges for companies in the context of continuous experimentation that lack both research and appropriate tooling.

Motivating Example

AB Inc[1] hosts an e-commerce platform offering handicraft products. The platform's application architecture consists of multiple microservices independently developed and released by small, autonomous teams [Newman, 2015]. These services include customer-facing frontend services (e.g., *landing page*, *product* catalog, *search*), and multiple business-related services (e.g., *accounting*, *shipping*). To keep pace with competitors the company decides to work on a *recommendation* feature serving the platform's users custom product suggestions based on their own and other users' search and purchase histories. The new recommendation feature requires changes on multiple services, thus, after consultation with the respective team leaders, the responsible release engineer decides to conduct an experiment to keep the risks of the change small, and ensure that the daily business is not affected. Consequently, the recommendation feature is tested on a small fraction of the user base while the majority of the users continues to use the application as usual. The experiment should confirm (1) the new service's functionality (e.g., scaling capabilities) and (2) the user's acceptance of the feature. This imposes multiple challenges for the release engineer:

Experiment Planning: Ultimately, the goal is to design experiments such that enough data points are collected to guarantee valid statistical conclusions (cf. Kohavi et al. [2014]). This involves decisions on when to start, for how long to execute, and which users to assign to the *recommendation* experiment. These decisions are aggravated when running multiple experiments in parallel, as the case for AB Inc. Overlapping experiments such as users being part of the *recommendation* experiment and a conflicting experiment testing a new *landing page* layout have to be avoided.

Experiment Execution: Running the *recommendation* experiment requires assessing the new service's scalability when exposed to a gradually increasing number of users, and in case of positive trends on collected metrics, an A/B test to measure the new feature's business impact. Manually administering multiple experiments running in parallel, involving keeping track of their execution

[1]The story, all names, and the company portrayed in this example are fictitious.

states (i.e., which users are when assigned to which experiments), consistently monitoring their key metrics, and triggering actions in case of spotted deviations, is challenging and prone to errors (e.g., misconfigured user assignments).

Experiment Analysis: Continuously assessing the health states of (running) experiments is crucial. The release engineer has to decide upon experiment continuation and cancellation. For example, whether to expose the recommendation feature to 10% of the users, or whether to launch the subsequent A/B test. Running multiple experiments in parallel bears the risk that malfunctioning or misconfigured experiments cause cascading, yet measureable effects that could, if interpreted incorrectly, ultimately lead to the termination of the wrong experiments. Consequently, this could hamper the evolution of entire services and, in the worst case, negatively impact business success when promising features are discarded only because of misinterpreted metrics and health states.

Goal

In this work we address these challenges that map to the life cycle phases of continuous experiments: planning, execution, and analysis. The hereby gained insights and resulting approaches should assist developers and release engineers when conducting continuous experiments. This is also reflected in the following **thesis statement**:

> ***A detailed understanding of the characteristics of continuous experiments enables building a conceptual framework for planning, executing, and analyzing experiments.***

We first need a proper understanding of continuous experimentation in both the industrial and research context. For this purpose, we conducted an in-depth, empirical assessment of the current state of continuous experimentation. The insights gained within this study built the basis for the development of research prototypes and approaches forming a conceptual framework. This framework **enables**:

1. the creation of **valid and optimized schedules** when planning experiments, which ensures the collection of enough data points fostering the

validity of statistical conclusions while at the same time fulfilling domain-specific constraints,
2. the **adherence to the specification** during experiment execution, including conditional chaining and fallback states to automatically react to abnormalities for keeping the impact of failing experiments low, and
3. the **identification of changes** in the context of experiments and **reasoning** about their effects on the application's health state during experiment analysis.

To pursue the goal in this thesis, we will focus on three research questions that are formulated in Section 1.1. The approach and main results of our research are presented in Section 1.2. Detailed background information and related work is covered in Section 1.3. The limitations of our approach are discussed in Section 1.4, followed by scientific implications in Section 1.5 and potential future work in Section 1.6. We summarize the contributions of our work in Section 1.7, and provide a roadmap of this thesis in Section 1.8.

1.1 Research Questions

The objective of this research is to support developers and release engineers in conducting continuous experiments. To gain a better understanding of the problem domain and properly investigate ways to satisfy the thesis statement, three research questions are explored. Figure 1.1 illustrates these research questions and how they are connected. To answer them, we conduct an empirical study to learn about the state of practice, and informed by these findings, we develop multiple research approaches and prototypes.

1.1.1 Understanding Continuous Experimentation

Staying ahead of competition requires a deep understanding of the field and all of its subtleties. This 'wisdom' is not only integral for data-driven decision making in the context of experimentation, but applies to research as well: to improve, one must understand. Consequently, to support developers and release engineers

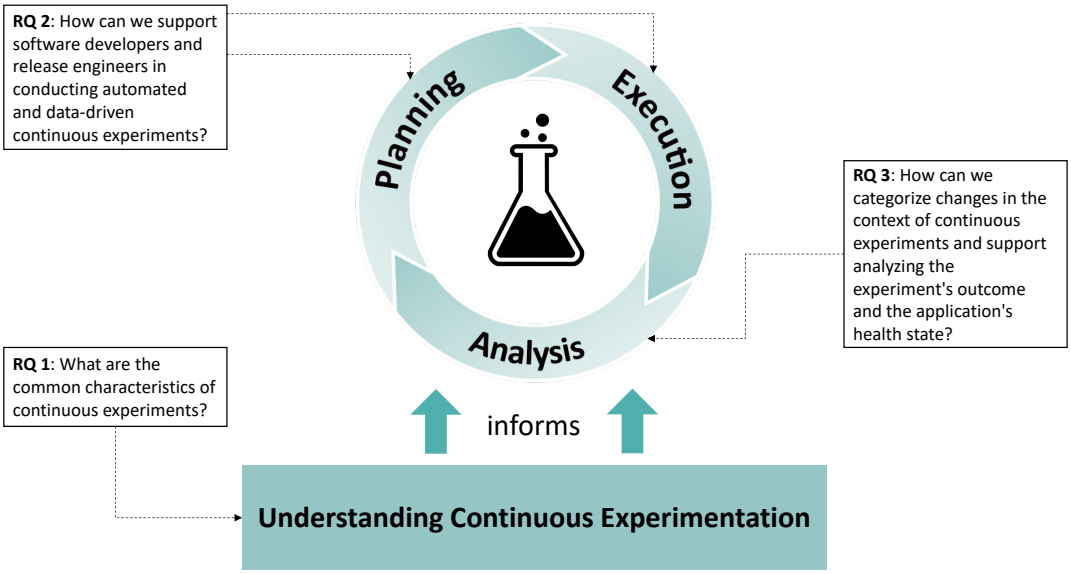

Figure 1.1: Overview of Research Questions.

throughout the life cycle of experiments we need to get a proper understanding of the state of practice. In recent years, there have been a multitude of studies in this field. This ranges from reports from a data science perspective (e.g., Kohavi et al. [2013, 2014]; Tang et al. [2010]), to reports from the software engineering angle (e.g., Fabijan et al. [2017]; Kevic et al. [2017]. All of these studies have in common that they mainly represent experience reports of single companies. In our work, we want to shed light on release processes of companies across multiple domains and of various sizes:

RQ 1: What are the common characteristics of continuous experiments?

We conducted an exploratory, empirical study involving both qualitative and quantitative phases to assess the current state of continuous experimentation. The findings of this study not only influenced large parts of this dissertation but also built the basis for our conceptual framework for experimentation involving the development of new research approaches and prototypes.

We learned, amongst others, that experimentation is especially enabled by architectures that foster independently deployable services. However, these architectures impose proper solutions (1) to manage the state of experiments, and (2) to assess the health states of experiments and the entire application in the context of experiments running in parallel. We will investigate the former in RQ 2, while we focus on health assessment in RQ 3.

1.1.2 Planning and Executing Experiments

In RQ 2, we aim to directly support developers and release engineers in planning and executing experiments. This involves fostering the parallel execution of experiments in microservice-based applications with as little overhead as possible, identifying optimal ways to collect required sample sizes (cf. Kohavi et al. [2014]) in minimum time, and making sure that experiments do not negatively influence each other. Hence, RQ 2 is formulated as follows:

> **RQ 2:** How can we support software developers and release engineers in conducting automated and data-driven continuous experiments?

We developed two research approaches that have been validated through prototyping and numerical experimentation. The first approach targets the planning phase of the experiment life cycle and scheduling in particular. The second approach supports the execution phase of the life cycle and involves the automated, data-driven execution of multi-phased experiments (e.g., a canary release automatically followed by an A/B test as the case in the AB Inc example).

Regarding the former, our evaluation has shown that our proposed implementation Fenrir outperforms other approaches not only in the quality of the experiment schedules identified but also in terms of execution time. Regarding the latter, an evaluation on runtime behavior has shown that our research prototype Bifrost supports running more than a hundred experiments in parallel without a significant performance degradation. These promising results suggest that our approaches scale to real-life release engineering scenarios.

1.1.3 Assessing Experiments

In RQ 2 we tackled the automated, data-driven execution of experiments based on pre-specified health criteria, in RQ 3 we take a step back, acknowledging that a fully automated execution is not the ideal choice under all circumstances, especially when dealing with experiments involving multiple complex changes. Consequently, we want to make developers and release engineers aware of the changes in the context of experiments, and how likely are these changes affecting the experiments' outcomes and health states to support their decision process.

> **RQ 3:** How can we categorize changes in the context of continuous experiments and support analyzing the experiment's outcome and the application's health state?

Similar to RQ 2, we developed a research approach that was validated through prototyping and numerical experimentation. We identify and categorize changes in the context of experiments on a topological level, involving all services or microservices that are interacted with.

We extensively evaluated our approach on multiple release scenarios. The produced rankings – identified changes are ordered according to their potential impact on the experiments' health states and outcomes – achieve promising results for our evaluation criteria.

1.2 Approach and Main Results

To explore and answer our research questions, we conducted an empirical study to learn about the state of practice, and informed by these findings, we developed and extensively evaluated several research approaches. In the following, we describe our methodology and summarize findings of our research with respect to the research questions introduced in Section 1.1. Figure 1.2 gives an overview of how single approaches are connected with each other. More details about individual findings and underlying approaches are presented throughout Chapters 2–5.

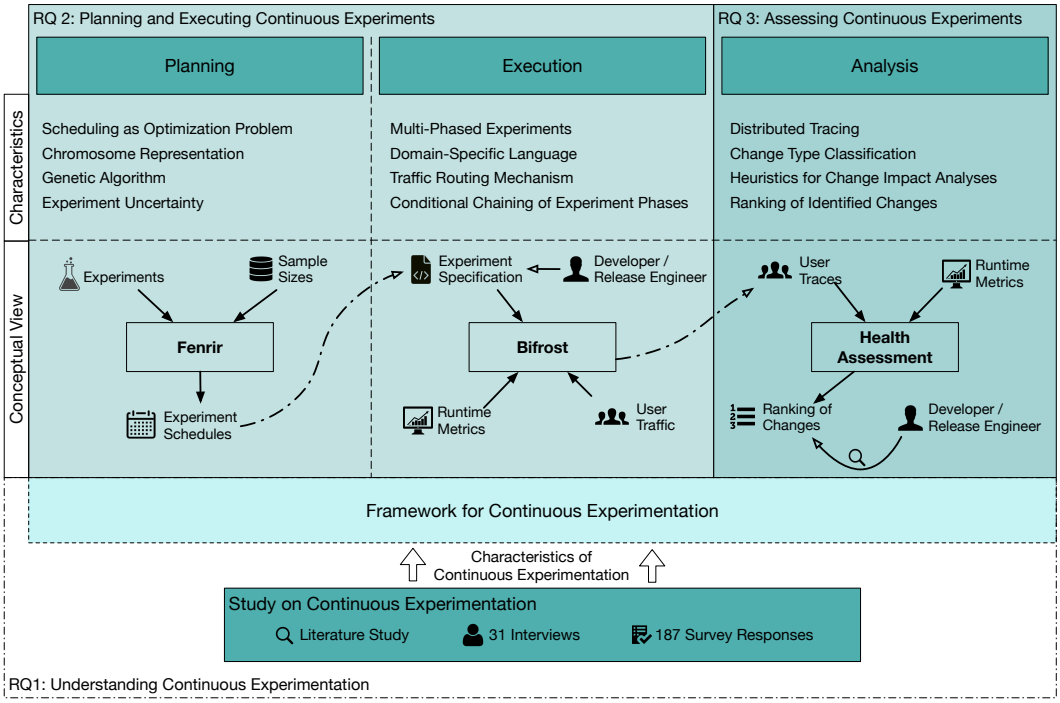

Figure 1.2: Overview of Research Approach.

1.2.1 Understanding Continuous Experimentation (RQ 1)

In the following, we describe our study's methodology and present selected key insights involving the *flavors of experimentation* and *enablers and obstacles* for experimentation. We conclude with a reflection on the *implications* the findings of our study have and present our *conceptual framework for continuous experimentation*.

Approach

For answering RQ 1, we conducted a mixed-method study [Shull et al., 2007] consisting of 31 semi-structured, qualitative interviews in two phases combined with a quantitative survey. Prior to conducting the first phase of interviews, we performed a literature study that served as a basis for formulating questions for the qualitative study. The first qualitative phase comprised 20 interviews

with developers or release engineers from companies across multiple domains and of various sizes. The interviews were structured into five themes: the release process in general, roles and responsibilities, quality assurance, issue handling, and release and experiment evaluation. We analyzed the qualitative data using open card sorting [Spencer, 2009] (683 cards in total), and categorized the participants' statements resulting in a set of findings. To validate and substantiate these findings on a larger sample size, we designed an anonymous Web-based survey consisting of 39 questions. In total, our survey attracted 187 complete responses (completion rate of 28% out of 667 responses). When revisiting our interview and survey findings, we identified three topics to be of particular interest: (1) experiment design (e.g., metrics, hypotheses, duration), (2) implementation techniques for experiments, and (3) experiment result interpretation. To get more profound insights, we defined a set of 32 more detailed questions and conducted a second qualitative phase involving 11 semi-structured interviews. Again, qualitative data was analyzed using open card sorting. For planning, conducting, and reporting our study, we followed the case study protocol proposed by Brereton et al. [2008]. We further considered the guidelines reported by Runeson et al. [2012]. Details on the design and execution of our study can be found in our case study protocol in Section 2.A.

Flavors of Experimentation

In our study, we learned that companies use different practices of continuous experimentation for different purposes. We introduced the terms *regression-driven* and *business-driven* experiments to classify these practices:

Regression-driven experiments are fundamentally a quality assurance technique used to identify whether changes have a negative impact on system properties, e.g., bugs, performance regressions, the application's scalability. Associated practices involve *canary releases* [Humble and Farley, 2010], *dark launches* [Feitelson et al., 2013; Tang et al., 2015], and *gradual rollouts* [Humble and Farley, 2010]. Our survey has shown that 37% of the participants make use of regression-driven experimentation practices. Experiments typically run from minutes to multiple days, are small scoped (i.e., small percentages of

users, user groups, or regions), and involve the collection of multiple application and infrastructure-level metrics (e.g., response time), sometimes also simple-to-measure business metrics. Interestingly, data interpretation is often driven by intuition and based on "gut-feeling" rather than on rigorous formal processes.

Business-driven experiments guide different implementation decisions or variants of features from a business perspective (e.g., do customers appreciate a feature). The most important type of this class is A/B testing [Kohavi et al., 2013], sometimes also referred to as controlled experiments. According to our survey participants, 23% are using this practice for testing new features. Experiments typically run for multiple weeks or even months, involve two or more user groups (or regions) of constant size, and primarily involve the collection of specific business metrics, sometimes in combination with a small selection of "key application-level" metrics. Data interpretation is characterized through rigorous hypothesis testing on selected metrics.

Enablers and Obstacles

During our study, we further learned what principles and practices enable and hinder organizations to leverage continuous experimentation:

Architecture. A suitable software architecture has been shown to be essential for experimentation. We identified a trend towards smaller, independently deployable services (i.e., microservices) [Newman, 2015]. Legacy system architectures turned out to be a dividing barrier between companies that do and those that do not adopt experimentation as part of their release process. Our study participants reported not suitable architectures as the main reason against conducting experiments (57% for regression-driven, 50% for business-driven experiments).

Data Science. Running experiments is all about data-driven decision making. If there are not enough users to experiment with, it becomes challenging to ensure statistical validity. 39% of our survey participants stated that they simply do not have enough users to conduct regression-driven experiments, while 28% stated that this is an issue for business-driven experiments. Related challenges, especially surfacing in smaller companies, are limited expertise to set up a suitable

experimentation infrastructure and the lack of data science knowledge to correctly interpret collected data.

Domain and Costs. Obstacles do not always have a sole technical origin. 39% percent of the participants stated that for the domain they are operating in, it does not make much sense to conduct experiments, or even the slightest regressions on a subset of the user base is unacceptable, e.g., in highly regulated or sensitive domains such as finance or health care. In the case of business-driven experimentation, 53% of the respondents stated that it is a business decision to not run experiments. Business owners have decided that setting up an experimentation infrastructure is not worth the investments.

Implications

We briefly reflect on implications the findings of our study have. This includes implications that directly influenced the research conducted in this dissertation, and implications that yet need to be addressed and highlight gaps for future research.

Escaping Feature Toggles. We observed that feature toggles (applied by 36% of our survey participants) as a mechanism to circumvent architectural limitations come at the price of increased complexity, negatively affecting source code maintainability and code comprehension. To escape these downsides and the potentially resulting technical debt due to feature toggle usage as reported by Rahman et al. [2016], our tooling to support the execution of experiments (see RQ 2) is based on a runtime traffic routing mechanism. Runtime traffic routing was the second most mentioned experiment implementation technique across our survey participants (30%) and allows migrating the experimentation logic (i.e., controlling which users are allowed to access experimental features) from source code to the network level. Consequently, services are treated as black boxes, promoting the usage of immutable deployments [Cito et al., 2015a] and following the mindset of cloud-based software engineering: service instances are volatile, they come and go, which is even aggravated in the context of experimentation.

Experiment Health Assessment. While microservice-based architectures emerged as key enabler for continuous experimentation, in our study, we also

learned that strictly following this architectural style is no silver bullet. It requires careful design decisions (e.g., where to host and persist data), but we observed that practitioners currently lack means to decompose an application into microservices in the first place. Moreover, in contrast to local function calls, as the case for monoliths, service interactions are subject to network failures and fluctuations, and identifying issues and their root causes across an entire network of services is challenging as mentioned by our interview participants. The latter becomes even more complex if multiple experiments are conducted by independent teams on various parts of the system. In RQ 3 we explored this challenge in greater depth.

From Intuition to Principled Decision Making. We observed that many developers and release engineers follow intuition-driven and experience-based approaches when defining metrics and thresholds to evaluate the success of regression-driven experiments. With our research approach and prototype Bifrost (presented in RQ 2), we envision to evolve more towards well-defined, structured experimentation processes. Experiments consist of multiple phases, each phase specifies health criteria (i.e., selected key metrics and their thresholds) to be fulfilled and actions in case of positive or negative evaluation outcomes. In our study, we also learned that the selection of (1) features to experiment with and (2) the fraction of the user base is rarely based on sound statistical or empirical evidence. Hence, research should strive to identify, for various application types, the principal metrics that allow for evaluating the success of an experiment, and identify best practices (e.g., in form of guidelines) or approaches on how to select changes that require experimentation. This is especially relevant as we have seen that developers and release engineers cannot generally be expected to be trained data scientists.

Conceptual Framework for Experimentation

In our study, we learned about characteristics of continuous experiments. During the course of research conducted within this dissertation we developed multiple approaches and tools that in combination form our framework for experimentation. Each of these approaches is built on an underlying (formal) model. These models

serve not only as a basis for the implementation of research prototypes but also for potential future extensions. Our framework involves a *planning* model for scheduling experiments, an *execution* model for executing experiments in a data-driven manner, and an *analysis* model for experiment health assessment. We will summarize key aspects of these models in the following, detailed descriptions can be found throughout Chapters 2–5.

Planning model. This model is designed to support finding solutions for an optimization problem, in our case finding valid schedules with maximal fitness for executing experiments. Maximal fitness means collecting the required sample sizes for all experiments in minimum time while ensuring that experiments do not overlap. This model is characterized by a chromosome representation of the problem, which comprises all experiments and their individual execution plans. A detailed description of the problem representation, thus the planning model, is given in Chapter 3.

Execution model. This model covers all the services and the users being part of experiments and formally maps to a state machine. States represent specific user assignments, e.g., which users are assigned to the *recommendation* service. In each state, a set of so-called *checks* is executed ensuring that the services under experimentation behave as expected. The outcome of checks then determines the subsequent state. This might include "fallback" states to immediately react to encountered problems, e.g., reassign all users to the stable, previous version with the *recommendation* functionality turned off. This detailed representation of states and transitions allows us to combine and chain multiple experimentation practices to form multi-phased continuous experiments. Chapter 4 describes the execution model in detail.

Analysis model. This model focuses on service interactions in the context of experiments. It is characterized by multiple service interaction graphs, in which nodes denote endpoints of services in specific versions and edges the interactions between them, thus which services call which concrete other service endpoints. Comparisons of such interaction graphs in the context of experiments reveal changes (e.g., added or removed services) on a topological level. Further details on this model are described in Chapter 5.

1.2.2 Planning Experiments (RQ 2)

In the following, we focus on the planning aspect of research question 2. We provide details on how we approached scheduling continuous experiments and briefly reflect upon our insights from developing and assessing our research prototype.

Approach

As mentioned in Section 1.1.2, we developed two research approaches that led to prototypes for experiment scheduling and execution: Fenrir and Bifrost. Fenrir is built on top of our planning model and is designed to schedule continuous experiments by taking uncertainty into account. In this context, uncertainty means that, until tested in production environments, companies do not know whether users appreciate new features, or in general, how changes associated to experiments affect the application's health state and a company's business success. Consequently, experiments are prone to change, get canceled frequently (i.e., freeing pre-scheduled resources), or are adjusted and restarted, and new experiments are added regularly. We formulate experiment scheduling as an optimization problem with the aim of fostering the parallel execution of experiments while at the same time ensuring that enough data is collected for every experiment to avoid overlapping experiments. The objectives of our optimization problem are (1) *duration*, as experiments should not last longer than needed, (2) *start time*, as experiments should start as soon as possible, and (3) *user group coverage*, as new features should be tested on preferred user groups if specified. Further, schedules are valid if and only if they fulfill a set of constraints. We distinguish between *experiment constraints* (e.g., non-interrupted experiments, reaching the minimum sample size) and *overarching constraints* (e.g., do not schedule more resources than available). We propose a genetic algorithm that operates on top of our chromosome representation of the optimization problem. For the evaluation, we compare it with other common search-based approaches: random sampling, local search, and simulated annealing. We assess the capabilities of these four algorithms in three aspects: (1) maximum fitness

(i.e., a measure to quantify a schedule's quality) scored for a specific set of experiments to be scheduled, (2) scheduling an increasing number of experiments at the same time, and (3) dealing with the reevaluation of existing schedules.

Scheduling Experiments with Fenrir

Fenrir was developed to produce a valid experimentation schedule with maximal fitness. With respect to our evaluation, our results are very promising, especially when we envision scheduling and re-scheduling of already running experiments to become an active part in a release pipeline, e.g., scheduling is triggered as soon as source code changes pass the quality assurance phases. We found that when it comes to larger number of experiments (≥ 20) to schedule, the genetic algorithm (GA) not only outperforms the other approaches in the fitness scores, but also drastically in execution time. For example, when scheduling 40 experiments with high required sample sizes, the GA reaches 62% of the maximal fitness score, simulated annealing (SA) 42%, and local search (LS) 43%. While it takes the GA on average 110 minutes to schedule these experiments, LS and SA take almost three times as long on average (280 and 274 minutes). Our evaluation was conducted on low-end public cloud instance types, and due to the nature of the genetic algorithm we assume that even higher parallelization is possible. Thus, with stronger computing machinery and more sophisticated parallelization strategies we expect that we can drastically decrease the time needed to find suitable solutions, a crucial factor for companies with high deployment frequencies. We also believe that there is space for improvement when it comes to the genetic algorithm's crossover strategy, which controls how an offspring of two parent individuals (i.e., valid schedules) is created. During our evaluation, we identified that our rather simple strategy of combining individuals leads to many invalid schedules.

Furthermore, our evaluation demonstrated that our approach is capable of reevaluating existing schedules, i.e., taking into account experiments that (1) finished within the already executed period, (2) got canceled, and (3) are added to be scheduled as well. The gap between fitness scores of the schedules produced by the various algorithms is smaller in these cases. The reason is that

SA and LS benefit from a highly optimized schedule to be reevaluated as the initial schedule was created by the genetic algorithm.

1.2.3 Executing Experiments (RQ 2)

In the following, we present our approach and findings for the second aspect of research question 2: executing continuous experiments.

Approach

We developed Bifrost, a middleware designed to support the automated and parallel execution of multi-phased continuous experiments in microservice-based applications. It builds upon a traffic routing mechanism to bypass the downsides inherent to feature toggles, which are reported as a potential source for technical debt [Rahman et al., 2016]. Experiments often comprise multiple phases. For example, AB Inc canary releasing to a subset of the users followed by a gradual rollout exposing the new feature to more and more users as long as defined health criteria are met, and when the new recommendation feature proved to be stable, an A/B test follows to assess user acceptance. Bifrost is designed to support conditionally chaining such experiment phases of various experimentation types (e.g., dark launches, canaries, gradual rollouts, A/B tests). Conditional chaining allows triggering automated actions such as rollbacks in case of spotted irregularities or enacting the re-execution of an experiment phase in case not enough data was collected. To serve reuse and easier comprehension, in Bifrost, experiments (and their single phases) are specified in a domain-specific language, a principle that we refer to *experimentation-as-code*. We extensively evaluate Bifrost in concrete scenarios of a microservice-based case study application to assess its capability to orchestrate experiments in large-scale applications consisting of hundreds of distributed services. This involves evaluating (1) the performance overhead introduced to systems when Bifrost is deployed, and identifying Bifrost's scaling capabilities when confronted with (2) a large number of multi-phase experiments executed in parallel and (3) experiments with a large set of continuously evaluated metrics and health checks.

Executing Experiments with Bifrost

Even though our experiments were conducted on low-end public cloud instances, Bifrost adds on average only 8 ms performance overhead when executing a four-phase experiment in comparison to a baseline application without Bifrost deployed. These four phases included a canary release to start off, followed by a dark launch assessing scalability, an A/B test to assess two alternative implementations from a business perspective, and a final gradual rollout, exposing the "winning" alternative in a stepwise manner to all users.

Impact of Experimentation Practices. We learned that dark launching requires a certain level of caution, its underlying traffic duplication might drastically increase load in parts of the system if the services under test involve outgoing calls, triggering cascading effects. Regarding A/B testing, we observed exactly the opposite. Traffic is split between alternative implementations leading to load-balancing effects, which reduces the measured overhead of our middleware down to 4ms.

Scaling Capabilities. We observed that our concept and prototype implementation is able to scale to very high numbers of parallel experiments and their involved health checks. When considering that even industry leaders in continuous deployment, such as Facebook [Savor et al., 2016; Tang et al., 2015], deploy between 100 and 1000 times a day, this is a good indication that our middleware is able to handle realistic concurrent deployment numbers even on low-end public cloud resources. Moreover, our prototype implementation based on Node.js is not optimized for performance, and a more efficient implementation would likely be feasible.

Experimentation-as-Code. Besides promising results demonstrated in our numerical evaluation, our approach offers multiple advantages. Most importantly, formalizing experiments in a domain-specific language, a principle we refer to as *experimentation-as-code*, fosters transparency, and allows experiments and their phases to be shared, reused, and versioned. Further, our approach is based on a formal model for executing continuous experiments. This allows us to investigate ways to build tools for experiment verification and validation. Assessing the health state of experiments in RQ 3 was a first step in this direction.

1.2.4 Assessing Experiments (RQ 3)

In the following, we will briefly reflect on our approach and main results for assessing continuous experiments (RQ 3).

Approach

We developed a research prototype that is based on our analysis model and considers changes in the context of experiments by analyzing distributed traces (as produced by Zipkin [2019] or Jaeger [2019]) of services interacting with each other. The addition, removal, or version updates of services are reflected in those traces, which enables us to identify changes on the topological level when comparing user traces of experimental and baseline (i.e., the stable variant of the application) versions of the application. During the scope of the change analysis we revisit the concept of uncertainty. Uncertainty not only plays an important role for canceled or adjusted experiment schedules but also when classifying changes. Changing only the internals of a service's implementation, without affecting the way it is consumed and the outgoing calls to other services it initiates, introduces less uncertainty than deploying and consuming a completely new service. Our approach incorporates this notion and led to the development of three concrete heuristics to produce a ranking of changes based on their potential negative impact on the experiment's and application's health states. These heuristics involve an analysis of the complexity of the network of services interacting with each other (*subtree complexity heuristic*), a simple root cause analysis for spotting cascading effects (*response time analysis heuristic*), and a combination thereof (*hybrid heuristic*). The heuristics' implementations focus on changes with negative impact. Taking into account individual changes that cause positive effects, for example on response times, is a topic for potential future work. We evaluate our approach in two aspects: (1) an evaluation of the quality of the rankings produced by the heuristics, and (2) a performance evaluation assessing the execution behavior of our approach. Regarding the former, we assess the rankings produced on two concrete release scenarios involving the running case study application and when dealing with multiple breaking changes. This

ranking quality evaluation is based on nDCG (normalized discounted cumulative gain) [Järvelin and Kekäläinen, 2002], a well-established metric in the field of information retrieval. The performance evaluation focuses on the heuristics' execution behavior. We are specifically interested in how the heuristics perform when dealing with interaction graphs of various characteristics, e.g., number of endpoints, *deep* vs. *broad* graphs, and the number of changes.

Findings

We characterized typical change types that surface in the evolution of microservice-based applications. We distinguish two categories of change types: *fundamental* and *composed*. Exact formal definitions of these change types and examples are provided in Section 5.4.3.

- Fundamental change types:
  - Calling a New Endpoint
  - Calling an Existing Endpoint
  - Removing a Service Call
- *Composed* change types are a combination of *fundamental* change types and involve:
  - Updated Caller Version
  - Updated Callee Version
  - Updated Version

The heuristics we developed take these change types and our concept of uncertainty into account when producing a ranking of identified changes. In total, we assessed the quality of the produced rankings of 6 variations of three heuristics across two evaluation scenarios. For every scenario, we distinguish cases with and without introduced performance degradation.

Overall, the achieved results are very promising. Combining nDCG scores across all scenarios yields the highest (average) score of 0.94 on a scale from 0.0 to 1.0 for a *hybrid* heuristic which combines features of (1) the *subtree complexity* heuristic taking into account the structure of the interaction graphs, and (2) the *response time analysis* heuristic conducting a simple root cause analysis. We identified that, despite being the top heuristic on average, one variation of the

hybrid heuristics does not perform best for cases without performance issues. This is an indication that it would make sense to let developers or release engineers using our tooling toggle between multiple selected heuristics, which provide insights onto the application's state from different angles.

With respect to the performance evaluation, the execution times of all variations of the heuristics are promising. For example, service networks consisting of up to 10,000 endpoints (e.g., 1,000 microservices with 10 endpoints each) can be analyzed within 5 seconds, graphs with up to 4,000 endpoints within 1 second. Detailed analyses revealed that the execution times of the heuristics are very stable and that the "change frequency" of a graph, i.e., the extent of changes between the compared variants, does not influence their performance.

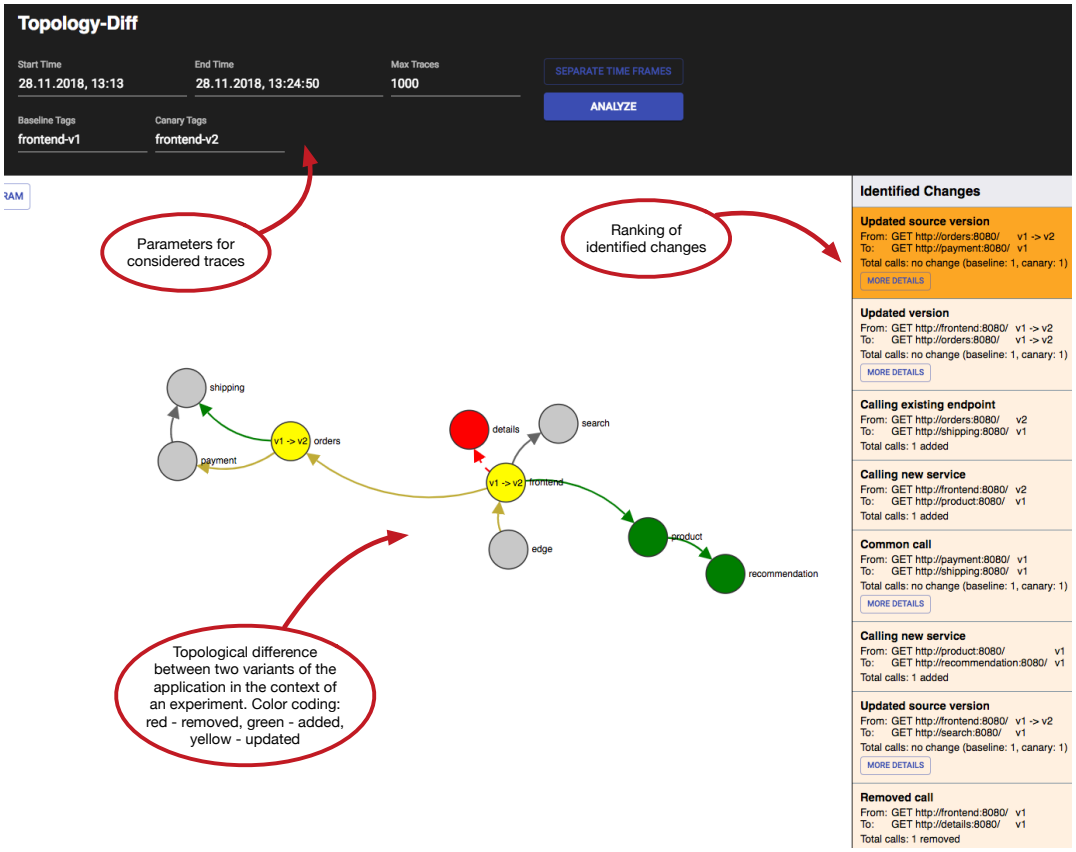

Figure 1.3: Visualization of identified changes in the context of an experiment and the ranking produced by one of the heuristics.

Our implemented tool chain, including a user interface visualizing the topological differences interactively (see Figure 1.3), complemented with the ranking of identified changes, is a valuable resource for developers and release engineers. It lets developers quickly get an overview of all the changes that are in place in the context of experiments. Moreover, it makes developers aware of erroneous or misconfigured deployments or experiments, e.g., misconfigured experiment routes are easier to spot if seen on the topological level rather than in log files distributed across multiple service instances.

1.3 Background and Related Work

We briefly review work related to our research that can be broadly categorized in (empirical) studies on continuous experimentation, conducting continuous experiments, and assessing experiments. Chapters 2 to 5 of this dissertation additionally cover related work that is specific to the research elaborated in the respective chapter.

1.3.1 Studies on Continuous Experimentation

There have been experience reports specifically investigating the process, challenges, and characteristics of conducting experiments in an enterprise setting. Fabijan et al. [2017] investigated the evolution of experimentation at Microsoft and presented a model detailing technical, organizational, and business evolution to provide companies a guidance towards data-driven product development. In more recent work, Fabijan et al. [2018] presented the A/B testing life cycle in place at Microsoft and discussed challenges and mitigation strategies. Also at Microsoft, but focusing on Bing as a case study, Kevic et al. [2017] analyzed more than 20,000 experiments conducted since 2014 and characterized the experimentation process by combining experiment data with source code artifacts. Results show, amongst others, that code changes for experiments are four times larger than other code changes. Lindgren and Münch [2016] were among the first empirically analyzing the state of experimentation by conducting a study with 10

Finnish IT companies. They came to the conclusion that it is not yet mature, as experimentation is rarely systematic and continuous. Success factors include deep customer and domain knowledge, and the availability of relevant skills (e.g., data science) and tools to conduct experiments. Their findings are similar to what we have learned in our study. Within the context of continuous experimentation, but more from the data science perspective is the work of Kohavi et al. [2013, 2014, 2009] and Crook et al. [2009]. These involve rules of thumb (e.g., collecting enough data is essential for statistical validity) and pitfalls to avoid (e.g., not filtering out Web-based robots) when conducting continuous experiments.

Different to these works, we empirically analyze the release processes and experimentation practices of companies from multiple domains and of varying sizes. Our approach is similar to Lindgren and Münch [2016]; however, we specifically focus on practitioners rather than on managers to gain insights not only on the process level, but also on technologies and environments. Moreover, we were also able to recruit several companies across multiple countries which make heavy use of experimentation.

1.3.2 Conducting Continuous Experiments

The challenge of dealing with multiple experiments in parallel and ensuring that they do not negatively influence each other can be tackled from multiple directions. This involves planning (and scheduling) their execution in advance and relying on routing and configuration logic during their execution to adhere to those plans, introducing general constraints and restrictions for experiments, and sophisticated filter mechanisms applied during experiment execution. While our approach follows the strategy of upfront planning and controlled execution, LinkedIn [Xu et al., 2015] tolerates overlapping experiments by default as in most cases their experiments are restricted to the UI level and run on different parts of the system. In contrast, Google [Tang et al., 2010] operates a system of multiple schematic layers and domains to divide up the user space to avoid overlapping and conflicting experiments. However, they assume that there is always sufficient user interaction available to collect enough data across all layers and domains, which may be true for Google, but less so for other companies

with different patterns of user interaction and a smaller user base. Tang et al. [2015] give insights how Facebook manages multiple versions running in parallel (e.g., using A/B testing) with a sophisticated *configuration-as-code* approach in which configuration files are generated and compiled from high-level source code insteand of being administered and modified manually. On top of this configuration approach, Veeraraghavan et al. [2016] describe how Facebook uses a tool called *Kraken* to route live user traffic on various layers (i.e., region, server, service) to identify and resolve performance bottlenecks across their application ecosystem. Continuing with Facebook, though from a data science perspective, Bakshy and Frachtenberg [2015] presented work on statistical methods to identify performance regressions in distributed systems.

Considering search-based techniques in the context of experimentation, Tamburrelli and Margara [2014] formulate automated A/B testing as an optimization problem. They propose a framework that supports the generation of different software variants using aspect-oriented programming, the runtime evaluation of these variants, and the continuous evolution of the system by mapping A/B testing to a search-based SE problem. In our approach, we rely on search-based techniques for scheduling continuous experiments.

1.3.3 Assessing Continuous Experiments

Previous research on assessing the outcome of continuous experiments, and canary releases specifically [Davidovic and Beyer, 2018; Tarvo et al., 2015], considers the service under test in isolation. These tools and approaches involve the automated collection of service-level metrics and apply statistics to identify deviations. However, the fundamental principle in service-based applications is hereby ignored, services communicate with each other and these interactions affect the overall application behavior, and consequently, can skew the health assessments of experiments. In our work, we specifically looked at these interactions with surrounding services, considering topological changes by relying on distributed tracing data, as collected for example by Jaeger [2019] and Zipkin [2019]. Sambasivan et al. [2011] proposed an approach similar to ours, they consider distributed traces to diagnose performance regressions, distinguishing

between *structural* and *response-time* mutations. While Sambasivan et al. assume similar workloads for the involved variants, our approach is specifically designed for experimentation and supports health assessment also when only a small fraction of users is assigned to experimental variants. Moreover, our approach is more fine-grained, we compare traces at the endpoint, version, and service levels.

1.4 Scope of Work, Potential, and Limitations

During the course of my PhD we developed research approaches and prototypes that were mainly influenced by our study on the state of experimentation as well as related work. While presented concepts can be applied in a more general context, research prototypes are often subject to certain limitations, which we summarize in the following. Moreover, we briefly reflect upon the main threats to the validity of the empirical study and specify the scope of this work.

1.4.1 Scope of Work

This work focuses on continuous experimentation practices and makes use of real users' interactions with a system. This allows not only revealing problems that are hard to identify when testing systems solely based on assumed and mimicked user behavior, for example by using load testing techniques (e.g., [Menascé, 2002]), but also getting insights to how users appreciate new features from a business perspective. Furthermore, we target service or microservice-based applications, which are typically accessed by customers or users as web applications over the Internet or are consumed by other services over an API. While the approaches for scheduling and executing multi-phased experiments presented in this dissertation could, in theory, also work for monolithic application architectures, we have no studies or prototypes that would show feasibility or effectiveness outside the described scope. The presented approach for experiment health assessment is designed for service-based architectural styles. One of the main factors for targeting service-based models is that we benefit from sophisticated telemetry

and monitoring solutions, which are often provided out of the box by Cloud providers and other service frameworks such as Istio [2019].

1.4.2 Potential for Industrial Adoption

In the scope of this dissertation, three research approaches and prototypes were developed and extensively evaluated. Before we reflect upon their limitations in Section 1.4.3, we briefly discuss their potential for industrial adoption.

Scheduling Experiments with Fenrir. Out of the three research prototypes we developed, the technology readiness level (e.g., as determined for EU research projects [European Commission, 2014]) of Fenrir is the lowest. Consequently, transforming Fenrir to an "industry-ready" solution is most challenging. Experiment schedules and their constraints we rely on for identifying and optimizing them are highly company and domain specific. As discussed in Section 1.3, companies follow different approaches for dealing with overlapping experiments and how they determine statistical validity. Moving beyond a prototype would require to take these various (additional) constraints into account, such that developers or release engineers are able to configure how Fenrir is generating and optimizing experiment schedules. In Section 1.6.4, we envision a different approach for this issue. Experiment verification based on statistical models might be a solution to revoke some of these constraints, making our approach more applicable across multiple domains.

Executing Experiments with Bifrost. We observed that our prototype is able to scale to very high numbers of parallel experiments, which is a good indicator for a potential industrial adoption. We have also seen that concepts very similar to those we proposed with Bifrost have found their way into industry. For example, Istio [2019], initially developed by IBM and Google, applies the same traffic routing approach of having lightweight proxies placed in front of service instances. Similar to our domain-specific language, the specification how traffic is routed within Istio is YAML-based. However, our approach based on conditional chaining and dividing the experiment execution into multiple phases goes beyond what the current implementation of Istio is capable of.

Assessing Experiments. During an internship at IBM Research, I worked with the team driving the research behind Istio. This resulted in the prototype for experiment health assessment covered in RQ3. Due to its ties with Istio, many components on which our approach is based on are in industrial use. For example, we extract distributed traces captured by tools such as Jaeger [2019] or Zipkin [2019], which are deployed within Istio. However, to transform our prototype to a solution of industrial strength more research is required when it comes to (1) identifying the most suitable heuristics for various application scenarios and (2) optimizing the visual representation of the changes identified and their impact when dealing with large ecosystems consisting of hundreds of distributed services.

1.4.3 Limitations

In the following, we reflect on the limitations of our work. We distinguish between threats to the validity of the findings of our empirical study, and limitations in the context of our research approaches and prototypes.

Empirical Study on Continuous Experimentation

We briefly describe the main threats to the validity of our empirical study's findings with respect to individual study phases. A detailed description of these threats is provided in Section 2.4.

Study Setup. The interview guide we created and the selection of questions for the qualitative phases of the study might have led participants to answer towards our possibly biased notion of continuous deployment and experimentation. We mitigated this researcher bias by building a foundation of understanding on the topic that is based on a literature study conducted in advance considering both academic work and online articles of well-known industry representatives.

Qualitative Phases. To recruit study participants we applied snowball sampling [Atkinson and Flint, 2001]. A potential threat of this strategy is that it may suffer from community bias, since the first participants are prone to impact the overall sample. We addressed this threat by selecting study participants

purposefully, focusing on practitioners rather than managers and also reviewed their online profiles, especially for those participants which were suggested to us via snowballing. For the second qualitative phase (deep-dive interviews), we were especially interested in experimentation practices. We provided potential interview candidates upfront with a brief outline of the goals of our study. While this allowed us to filter for participants that could reveal additional insights, this also introduced a potential threat that they shared information based on what they thought we wanted to know (i.e., hypothesis guessing), or withheld information and opinions that they thought would be unpopular (i.e., evaluation apprehension) [Wohlin et al., 2000]. We mitigated these threats by assuring that both their answers and company affiliation would be anonymized. Further, a potential threat to our empirical findings is that our results are not generalizable beyond the subjects involved in the interviews. We mitigate this effect by employing a mixed-method study validating our interview findings in a more general context using a quantitative survey.

Quantitative Phase. To attract a high number of survey respondents we advertised the survey on social media. Since participation in online surveys is voluntary, it is likely that the survey has attracted a respondent demography with substantial interest and familiarity with continuous deployment and experimentation practices leading to self-selection bias.

Planning, Executing, and Analyzing Continuous Experiments

We briefly summarize limitations with respect to the chosen case study applications, evaluation environments, and algorithms.

Case Study Applications. The main limitation with respect to our research prototypes is that each individual evaluation was conducted on case study applications, which affects the generalizability of our findings. Our evaluation of the scheduling prototype Fenrir only relied on self-generated experiments, even though we created them based on knowledge gathered from various literature sources such as Kevic et al. [2017] or Fabijan et al. [2017] (e.g., duration of experiments) and applied a real world traffic profile. To mitigate this threat we created multiple scenarios involving experiments with low, medium, and high

required sample sizes and evaluated the implemented algorithms on different numbers of experiments to schedule. Similar concerns apply to Bifrost. We cannot eliminate the possibility that our approach will have a higher overhead or scale worse for applications other than our case study application, which was – while realistic – designed specifically for this evaluation and thus is not a real production application. In case of experiment health assessment (RQ 3), our evaluation was conducted on traces of self-generated release scenarios. We mitigated this threat by covering a broad range of scenarios involving multiple case study applications and introduced sub-scenarios involving simulated performance issues.

Evaluation Environments and Metrics. The evaluation of both Fenrir and Bifrost prototypes was conducted in virtualized environments, the Google Cloud Platform. As performance was one of the essential criteria for the tools' assessments, especially for Bifrost, it is possible that performance variations inherent to public clouds have influenced the results of our evaluation [Leitner and Cito, 2016]. To mitigate this threat, we have repeated the various evaluation runs multiple times and reported observed deviations. A further limitation regarding the experiment health assessment involves the usage of nDCG (normalized discounted cumulative gain) as a metric to determine the quality of the produced rankings. The metric requires a relevance classification of all changes identified on a scale from not relevant to highly relevant, which was conducted by the authors of the paper. Consequently, this classification has a direct effect on the resulting nDCG scores.

Algorithms and Calibration. For Fenrir and the experiment health assessment we heavily relied on algorithms. Not only the selection and design of these algorithms have impact on evaluation results, but also their implementations strongly rely on parameter settings. For example, population sizes for the genetic algorithms and number of studied generations in case of Fenrir, and assigning scalar values to our characterized change types to quantify uncertainty in case of the experiment health assessment. To mitigate these threats we performed various calibration runs with different parameter settings. However, there might

also exist other heuristics and algorithms that provide better results than those we implemented.

1.5 Scientific Implications

In the following, we briefly reflect on the overall scientific lessons learned. As we discussed the implications of the empirical study in Section 1.2.1, the presented lessons focus on research questions 2 and 3 and thus the approaches and tools devised to support planning, executing, and analyzing experiments.

1.5.1 Taming Uncertainty

Uncertainty was a reoccurring theme during the course of this dissertation. This involved scheduling in the context of frequently canceled and adjusted experiments, and classifying changes for experiment health assessment. For these two instantiations we had to take multiple decisions to "tame" uncertainty. The level of detail and thus the granularity we wanted to act upon was the essential criterion and affected space and time complexities of all our proposed approaches. For example, is it enough to re-evaluate schedules on a daily basis, or is scheduling a routine that needs hourly iterations, and should changes be considered on the level of individual service endpoints, or is it better to treat them in an aggregated way on the service level? Answering these questions requires trade-offs affecting the qualities of the produced schedules and rankings of changes, and the computational resources and the time it takes to produce them. Our research approaches can be seen as a small piece in a big mosaic trying to handle uncertainty. While our approaches proved to be suitable for our evaluation scenarios, there might be other – yet unexplored – approaches that could go beyond what we have covered. Research should strive for a better understanding of uncertainty in the context of experiments, considering uncertainty from many different levels of granularity. In Section 1.6.5, we propose a first step in that direction by envisioning an extension of our health assessment approach.

1.5.2 Single Points of Failure

Being it proxies or sidecar components that are continuously interacting with central authorities to update traffic routing information, or feature toggle libraries that are syncing with key/value stores to determine for a certain user which code blocks to execute next, those experimentation implementation techniques have in common that single points of failure are re-introduced to distributed systems. While there have been decades of research on distributed systems to determine safe routines to handle situations in which the prime synchronization points disappear from a network, little is known on how to deal with such situations in the context of experiments. While container orchestration systems such as Kubernetes [2019] alleviate the issue by re-launching failed containers, the problem of experiment state synchronization remains. Research should strive for identifying approaches on how to (1) safely stop experiments and bring everything back to stable versions and keeping the impact of the failure low also in the context (database) schema changes, (2) handle instant take-overs of experiment states from a failing component, and (3) devise mechanisms to distribute experiment state among multiple components that can bear single failing components.

1.5.3 To Experiment, or not to Experiment

Even when equipped with an armada of experimentation tools ranging from planning over execution to analysis, potentially enhanced with ideas shared in Section 1.6, there are still some issues that are remaining. First, it is hard to identify with which features to experiment. Experimenting with all changes, i.e., "every feature is an experiment" Parnin et al. [2017], could have some drawbacks. It could not only introduce issues caused by overlapping experiments, it also increases the effort for administering them. Someone needs to plan experiments, decide on sample sizes, effect sizes to measure, metrics to observe, and all the other ingredients that our tooling and approaches require as input. Established statistical formulas help to determine minimum sample sizes and minimum experiment durations. However, it is often still unclear what metrics

to observe. Applying machine learning techniques on previous experiments and their characteristics and parameters might be a first promising step to provide insights for experiments similar in scope and services involved. As uncertainty is omnipresent, research needs to develop practices that help also in those cases such as the deployment of completely new services for which uncertainty is too high to receive fruitful results with traditional machine learning techniques. Another issue when applying such techniques is that they are based on the assumption that metrics and parameters used for previous experiments are considered optimal. This raises the question how to rate the success of experiments. Research should identify whether success is only bound to technical (e.g., response time) and business metrics (e.g., conversion rate), or whether there should be different and new ways to assess experiments. Gained insights could then not only serve as a vehicle to advance machine learning in the context of experimentation but also be beneficial for the analysis phase in the experiment life cycle.

1.6 Opportunities and Future Work

We demonstrated that our framework for experimentation enables planning, executing, and analyzing continuous experiments. In the following, we point out and elaborate multiple opportunities for future work that emerge from our findings.

1.6.1 Enhancing the IDE

Moving towards DevOps practices often requires developers to juggle their core task of writing code against many other responsibilities, including operations support, release planning, conducting experiments, and data analysis. While the IDE was the central place for writing code in the past decades, these new tasks often involve command line tools and dealing with configuration code instrumenting build servers, static analysis tools, experimentation platforms, or infrastructure provisioning. Instead of having multiple tools and their dashboards to keep track of, we envision the IDE to again become the central place for

developing and deploying software. This involves investigating new ways of how we can combine different artifacts such as source code and instrumentation code (e.g., specifying which experimental features are tested on which users). One goal is to raise the user's awareness which parts of the source code are currently under experimentation, and establishing links to telemetry solutions and experiment health assessment platforms giving upfront information about the current state of experiments (e.g., positive/negative trends on key metrics).

1.6.2 Smart Experimentation Platforms

The advantage of feature toggles lies in their simplicity. It does not take much to create a basic experiment running two or more versions on the same service instance. However, the more prevalent feature toggles are, and source code and experimentation logic gets tangled up, the less maintainable and testable code gets. In a first step, we envision an approach that is capable of injecting experimentation logic into the source code during compilation or code interpretation. Concepts known from aspect-oriented programming or byte code manipulation could serve as a starting point. In a second step, we envision this injection to become smarter. Code injection could be instrumented by experiment execution engines or platforms that know exactly about the states of various experiments, how much traffic is available or users are active, and how resources are utilized. Based on multiple factors, these engines could then decide whether experiments are executed on a single instance using feature toggles injected into the source code, or whether these experimental versions should be "split up", migrated and deployed onto multiple service instances by relying on traffic routing mechanisms for better load distribution (e.g., to expose more users to the experiment). This would separate experiment definition (in domain-specific languages) from experiment execution for all implementation techniques and keep source code clean, maintainable, and testable.

1.6.3 Experimenting with Runtime Changes

A vision that goes beyond these ideas and combines improved IDE support and smarter experimentation is to automatically inject (live) code changes into a running application. A first approach could be based on dark launches, having a separate, but identical version of the current application running in parallel to which code changes are injected. Every production request is then also forwarded to this manipulated version. The developer would immediately see the effects of code changes within the IDE (e.g., deviations in performance metrics). Thus, this approach would guide development decisions while writing code and could identify performance bottlenecks immediately before affected code changes are even committed to version control systems.

1.6.4 Experiment Verification

Our formally defined individual models that build the core of our framework provide a basis to add support for experiment verification, i.e., to identify upfront whether a defined experiment could negatively interfere with other planned or currently running experiments. This could involve statistical models that revoke some of the constraints we introduced for scheduling experiments (e.g., having users being part of multiple overlapping experiments) and thus allow executing more experiments at the same time. Another approach would be to build upon knowledge from product-line research by modeling dependencies between the various versions of services and triggering alarms if experiments are defined that rely on deployments that violate such dependency constraints.

1.6.5 Revisiting Uncertainty

For assessing the health state of experiments we rely on the concept of uncertainty, e.g., newly deployed services introduce higher uncertainty than deploying a new version of an already existing service. We envision to extend the scope of this concept of uncertainty. This could include the technology stacks service instances are running on (e.g., using prototypical compiler features) or user groups as

some groups might have different interaction patterns (e.g., being more open to changes). This updated uncertainty concept could then lead to more detailed insights when assessing both health states and outcomes of experiments.

1.7 Summary and Contribution

In our initial motivating example, we raised challenges in the context of continuous experimentation that lack both research and appropriate tooling. In the scope of this dissertation, we addressed these challenges that map to the life cycle phases of continuous experiments, namely planning, execution, and analysis. The gained insights and resulting research approaches and prototypes support our thesis statement introduced in Section 1: *"A detailed understanding of the characteristics of continuous experiments enables building a conceptual framework for planning, executing, and analyzing experiments."* We demonstrated that this framework and its underlying individual models build a promising basis for developing research prototypes and approaches to assist developers and release engineers throughout all experiment life cycle phases. We extended our framework during the course of this dissertation (i.e., planning and analysis approaches) and demonstrated in Section 1.6 that there are multiple opportunities for future work. To summarize, our work makes the following contributions:

- we present insights from an empirical mixed-method study assessing the current state of continuous experimentation. We classified experimentation practices into regression-driven and business-driven continuous experimentation.
- derived from the findings of our empirical study, we present a conceptual framework for continuous experimentation involving individual (formal) models that enable planning, executing, and analyzing experiments.
- we present multiple research approaches as concrete instantiations of these individual models mapping to the experiment life cycle phases of planning, execution, and analysis.

- we validated our research approaches through prototyping and numerical experimentation. We extensively evaluated them on multiple realistic release scenarios and provide open source versions of our prototypes to foster extensibility and reproducibility.

1.8 Thesis Roadmap

The remainder of this dissertation consists of four chapters, each published at an internationally renowned, peer-reviewed conference or journal. Figure 1.4 provides an overview of research activities differentiated by the research questions presented in Section 1.1.

Figure 1.5 lists publications that emerged from my PhD, but are orthogonal to or complementing the core publications of this dissertation. We group them in *continuous experimentation* (i.e., complementing publications), *continuous integration, delivery, and deployment*, and *empirical studies of ecosystems*.

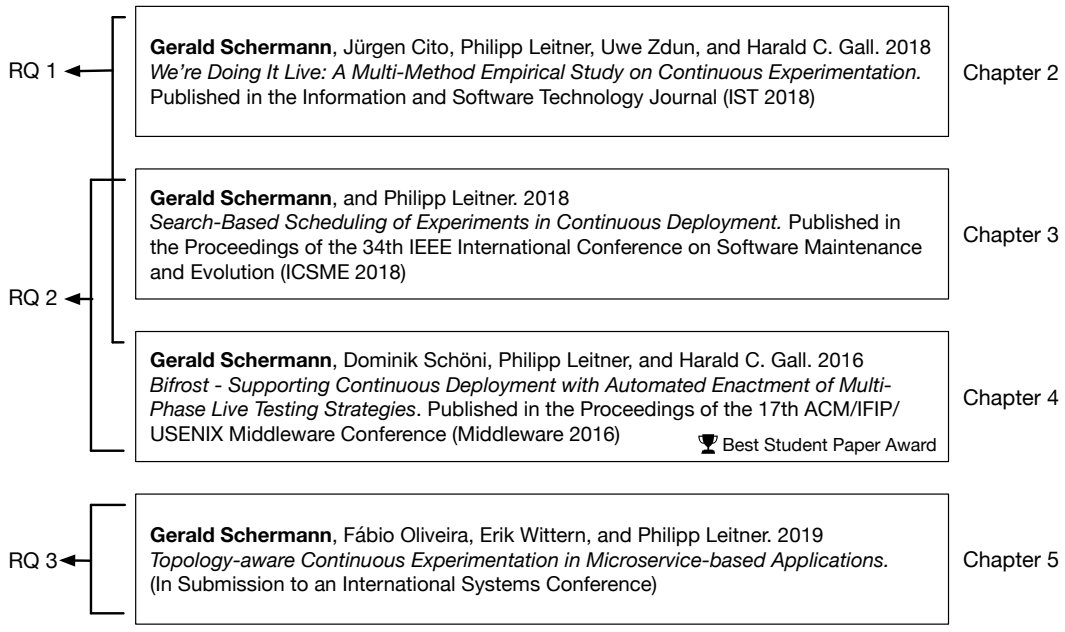

Figure 1.4: Roadmap of this dissertation relating research questions to publications.

Chapter 2 investigates the state of practice in continuous experimentation. This work was conducted in collaboration with my colleague Jürgen Cito, my supervisors Harald C. Gall and Philipp Leitner, and Uwe Zdun from University of Vienna.

Chapter 3 presents a search-based approach for scheduling continuous experiments. This work was done in collaboration with Philipp Leitner.

Chapter 4 presents a middleware for executing multi-phased continuous experiments. This work was done in collaboration with Dominik Schöni, a former student of University of Zurich who contributed with the development of the prototype, and my supervisors Harald C. Gall and Philipp Leitner. This chapter uses the term "live testing" synonym for "continuous experiment", the latter term is more established in recent literature and is used throughout the remainder of this dissertation.

Chapter 5 presents an approach for assessing health states of continuous experiments by considering topological changes. This work was created in collaboration with Philipp Leitner, and Erik Wittern and Fábio Oliveira from IBM Research.

Figure 1.5: Further publications that emerged from my PhD, but are not within the scope of this dissertation.

2

We're Doing It Live: A Multi-Method Empirical Study on Continuous Experimentation

Gerald Schermann, Jürgen Cito, Philipp Leitner, Uwe Zdun, Harald C. Gall
Published in the Journal of Information and Software Technology (2018)
Contribution: study design, data collection, data analysis, and paper writing

Abstract

Context: Continuous experimentation guides development activities based on data collected on a subset of online users on a new experimental version of the software. It includes practices such as canary releases, gradual rollouts, dark launches, or A/B testing.
Objective: Unfortunately, our knowledge of continuous experimentation is currently primarily based on well-known and outspoken industrial leaders. To assess the actual state of practice in continuous experimentation, we conducted a mixed-method empirical study.
Method: In our empirical study consisting of four steps, we interviewed 31 developers or release engineers, and performed a survey that attracted 187 complete responses. We analyzed the resulting data using statistical analysis and open coding.

Results: Our results lead to several conclusions: (1) from a software architecture perspective, continuous experimentation is especially enabled by architectures that foster independently deployable services, such as microservices-based architectures; (2) from a developer perspective, experiments require extensive monitoring and analytics to discover runtime problems, consequently leading to developer on call policies and influencing the role and skill sets required by developers; and (3) from a process perspective, many organizations conduct experiments based on intuition rather than clear guidelines and robust statistics. *Conclusion:* Our findings show that more principled and structured approaches for release decision making are needed, striving for highly automated, systematic, and data- and hypothesis-driven deployment and experimentation.

2.1 Introduction

Many software developing organizations are looking into ways to further speed up their release processes and to get their products to their customers faster [Chen, 2015]. One instance of this is the current industry trend to "move fast and break things", as made famous by Facebook [Feitelson et al., 2013] and in the meantime adopted by a number of other industry leaders [Rubin and Rinard, 2016]. Another example is continuous delivery and deployment (CD) [Humble and Farley, 2010]. *Continuous delivery* is a software development practice where software is built in such a way that it can be released to production at any time, supported by a high degree of automation [Fowler, 2013]. *Continuous deployment* goes one step further; software is released to production as soon as it is ready, i.e., passing all quality gates along the deployment pipeline. These practices pave the way for controlled continuous experimentation (e.g., A/B testing [Kohavi et al., 2013], canary releases [Humble and Farley, 2010]), which are a means to guide development activities based on data collected on a subset of online users on a new *experimental* version of the software. Unfortunately, our knowledge of continuous experimentation practices is currently primarily based on well-known and outspoken industrial leaders [Kohavi et al., 2013; Tang et al., 2010]. This is a cause for concern for two reasons. Firstly, it raises the question to what

extent our view of these practices is coined by the peculiarities and needs of a few innovation leaders, such as Microsoft, Facebook, or Google. Secondly, it is difficult to establish what the broader open research issues in the field are.

Hence, we conducted a mixed-method empirical study, in which we interviewed 31 software developers and release engineers from 27 companies. To get the perspective of a broader set of organizations, we specifically focused on a mix of different team and company sizes and domains. However, as continuous experimentation is especially amenable for Web-based applications, we primarily selected developers or release engineers from companies developing Web-based applications for our interviews. We combined the gathered qualitative interview data with an online survey, which attracted a total of 187 complete responses. The design of the study was guided by the following research questions.

***RQ1**: What principles and practices enable and hinder organizations to leverage continuous experimentation?*

We identified the preconditions for setting up and conducting continuous experiments. Continuous experimentation is facilitated through a high degree of deployment automation and the adoption of an architecture that enables independently deployable services (e.g., microservices-based architectures [Newman, 2015]). Important implementation techniques include *feature toggles* [Hodgson, 2016] and *runtime traffic routing* [Veeraraghavan et al., 2016]. Experimenting on live systems requires more insight into operational characteristics of these systems. This requires extensive monitoring and safety mechanisms at runtime. Developer on call policies are used as risk mitigation practices in an experimentation context. Experiment data collection and interpretation is essential. However, not all teams are staffed with experts in all relevant fields, we have seen that these teams can request support from internal consulting teams (e.g., data scientists, DevOps engineers, or performance engineers).

***RQ2**: What are the different flavors of continuous experimentation and how do they differ?*

Having insights into the enablers and hindrances of experimentation, we then investigated how companies make use of experimentation. Organizations use dif-

ferent flavors of continuous experimentation for different reasons. *Business-driven experiments* are used to evaluate new functionality from a business perspective, first and foremost using A/B testing [Kohavi et al., 2013]. *Regression-driven experiments* are used to evaluate non-functional aspects of a change in a production environment, i.e., validate that a change does not introduce an end user perceivable regression. In our study, we have observed differences in these two flavors concerning their main goals, evaluation metrics, how their data is interpreted, and who bears the responsibility for different experiments. We have also seen commonalities in how experiments are technically implemented and what their main obstacles of adoption are.

Based on the outcomes of our study, we propose a number of promising directions for future research. Given the importance of architecture for experimentation, we argue that further research is required on architectural styles that enable continuous experimentation. Further, we conclude that practitioners are in need of more principled approaches to release decision making (e.g., which features to conduct experiments on, or which metrics to evaluate).

The rest of this paper is structured as follows. In Section 2.2, we introduce common continuous experimentation practices. Related previous work is covered in Section 2.3. Section 2.4 gives more detail on our chosen research methodology, as well as on the demographics of our study participants and survey respondents. The main results of our research are summarized in Sections 2.5 and 2.6, while more details on the main implications and derived future research directions are given in Section 2.7. Finally, we conclude the paper in Section 2.8.

2.2 Background

Adopting CD, thus increasing release velocity, has been claimed to allow companies to take advantage of early customer feedback and faster time-to-market [Chen, 2015]. However, moving fast increases the risk of rolling out defective versions. While sophisticated test suits are often successful in catching functional problems in internal test environments, performance regressions are more likely to remain undetected, hitting surface only under production workloads [Foo et al.,

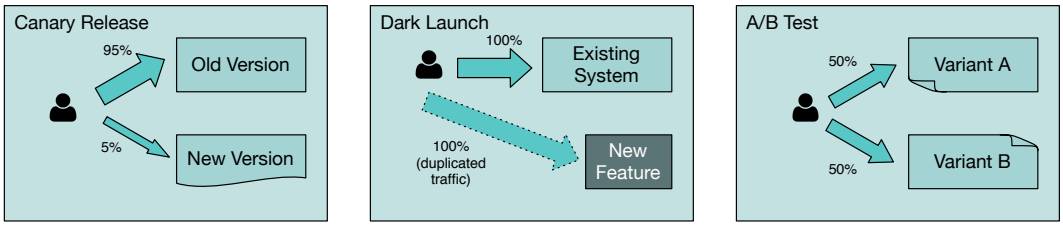

Figure 2.1: Overview of canary releases, dark launches, and A/B testing.

2015]. Techniques such as user acceptance testing help companies estimate how users appreciate new functionality. However, the scope of those tests is limited and allows no reasoning about the demand of larger populations. To mitigate these risks, companies have started to adopt various continuous experimentation practices, most importantly canary releases, gradual rollouts, dark launches, and A/B testing. We provide a brief overview of these experimentation practices in Section 2.2.1, followed by an introduction to two common techniques how these practices can be implemented in Section 2.2.2.

2.2.1 Experimentation Practices

Figure 2.1 illustrates the practices of canary releases, dark launches, and A/B testing.

Canary Releases. Canary releases [Humble and Farley, 2010] are a practice of releasing a new version or feature to a subset of customers only (e.g., randomly selecting 5% of all customers in a geographic region), while the remaining customers continue using the stable, previous version of the application. This type of testing new functionality in production limits the scope of problems if things go wrong with the new version.

Dark Launches. Dark, or shadow, launching [Feitelson et al., 2013; Tang et al., 2015] is a practice to mitigate performance or reliability issues of new or redesigned functionality when facing production-scale traffic. New functionality is deployed to production environments without being enabled or visible for any users. However, in the backend, "silent" queries generated based on production

traffic are forwarded to the "shadow" version. This provides insights into how the feature would be behaving in production, without actually impacting users.

Gradual Rollouts. Gradual rollouts [Humble and Farley, 2010] are often combined with other continuous experimentation practices, such as canary releases or dark launches. The number of users assigned to the newest version is gradually increased (e.g., increase traffic routed to the new version in 5% steps) until the previous version is completely replaced or a predefined threshold is reached.

A/B Testing. A/B testing [Kohavi et al., 2013] comprises running two or more variants of an application in parallel, which only differ in an isolated implementation detail. The goal is to statistically evaluate, usually based on business metrics (e.g., conversion rate), which of those versions performed better, or whether there was a statistically significant difference at all.

2.2.2 Implementation Techniques

The two common implementation techniques for conducting experiments are feature toggles and runtime traffic routing.

Feature Toggles. Feature toggles [Hodgson, 2016] are a code-level experimentation technique. In their simplest form, they are conditional statements in the source code deciding about which code block to execute next (e.g., whether a certain feature is enabled for a specific user or user group).

```
1 if isEnabled('newFeature', $user)
2     # code block containing new feature
3 else
4     # code block containing old functionality
5 end
```

Runtime Traffic Routing. Runtime traffic routing is a network-level experimentation technique. Multiple versions of an application or service run in parallel (e.g., as virtual machines, cloud instances, or containers). Depending on filter criteria applied on user requests (e.g., header information such as cookies, device identifiers), dynamically configured (network-level) components (e.g., proxies) decide to which concrete version of an application or service requests should be forwarded. A special type of traffic routing are blue/green

deployments [Bass et al., 2015], which include two or more active versions at the same time, but only one serves production traffic.

2.3 Related Work

Release engineering and CD is currently a popular topic of study in software engineering and data science. We categorized related work into (1) research related to continuous integration (CI) as a prerequisite for CD and continuous experimentation, (2) research related to CD including its adoption and challenges involved, and (3) research covering continuous experimentation practices and experience reports.

2.3.1 Continuous Integration

Continuous Integration (CI) as prerequisite for CD has been studied extensively in recent years. Vasilescu et al. [2015] studied the effects of CI in the context of open source projects that use pull requests on GitHub. Hilton et al. [2016] conducted a detailed analysis of the usage of CI in open source projects and showed that CI supports more frequent releases and is widely adopted by popular software projects. Recently, Hilton et al. [2017] reported on an empirical study investigating the barriers and needs developers face when using CI including trade-offs related to security, flexibility, and assurance. Similarly, Debbiche et al. [2014] reported on the challenges a telecommunications company faced on their way to adopt CI. Brandtner et al. [2014] have found that integrating build information from multiple sources across the CI tool chain can support developers to stay aware about the quality and health state of a software system. Ståhl and Bosch [2014] proposed a model for documenting the practice of CI derived from a systematic literature review and illustrated its application on an industry case study. In the scope of CI, there are also a multitude of research on software builds and testing. Beller et al. [2017] studied how central testing is to the CI process and analyzed more than 2 million builds on the Travis CI service. Similarly, Rausch et al. [2017] investigated the factors that lead to build failures

on Travis CI. A similar research question was also investigated by Vassallo et al. [2017], who additionally compared build failures of OSS projects with projects of a financial organization, leading to a taxonomy of build failures.

2.3.2 Continuous Delivery and Deployment

Roadmaps and Literature Reviews. Adams and McIntosh [2016] provided a roadmap for future research on CD and release engineering practices. Similarly, Rodríguez et al. [2016] conducted a systematic literature review on CD research articles and addressed potential fields for future research. In their systematic literature review, Shahin et al. [2017a] classified available approaches and tools in the context of CI and CD. Moreover, they identified challenges, practices, and gaps for future research considering the current state of CI and CD. Rahman et al. [2015] conducted a qualitative analysis of CD practices performed by 19 software companies by analyzing company blogs and similar online texts. However, they did not conduct interviews or a formal survey beyond what is already available in blogs. In their white paper, ThoughtWorks and Forrester Consulting [2013] conducted a survey with 325 business and IT executives and showed that many companies have a low level of maturity when it comes to CD, and consequently are not able to keep innovation as high as business aims for.

DevOps. There are also studies on the state of the art in DevOps. The most authoritative source on this comes from Puppet Labs [2016], a provider of Infrastructure-as-Code tooling, which releases annual reports on the state of DevOps. Academic studies in this field include our own previous work [Cito et al., 2015a] about integrating runtime monitoring data from production environments into developer tools, but also the work conducted in the CloudWave project [Bruneo et al., 2014]. Lwakatare et al. [2015] combined a literature survey and practitioner interviews to investigate the DevOps "phenomenon". They identified collaboration, automation, measurement, and monitoring as the characterizing DevOps elements. In a more recent study, Lwakatare et al. [2016] studied the relationship of DevOps to agile, lean, and CD approaches. Shahin et al. [2017b] identified different types of team structures by investigating how

development and operations teams are organized in the industry for adopting CD practices.

CD Adoption and Challenges. Recent research has comprised multiple studies on the challenges companies face when adopting CD. Leppanen et al. [2015] and Olsson et al. [2012] conducted studies with multiple companies discussing technical and organizational challenges, and their state of CD adoption. Similarly, Chen [2015], and Neely and Stolt [2013] provide experience reports from a perspective of a single case study company, the obstacles they needed to overcome and the benefits they gained by establishing CD-based release processes. Claps et al. [2015] identified social challenges that companies face, and present mitigation strategies. Recently, Chen [2017] presented six strategies to overcome adoption challenges and in addition proposed possible directions for future research. Bellomo et al. [2014] investigated architectural decisions companies take to enable CD and introduced deployability and design tactics. Itkonen et al. [2016] investigated the adoption of CD in a single case study company and report on the benefits it enables for both customers and developers. Fitzgerald and Stol [2014] reported on the need for a tighter collaboration between software development and business strategy to enable continuous planning. In previous work [Schermann et al., 2016a], we derived a model based on the trade-off between release confidence (i.e., the effort companies put into quality gates throughout the development process) and the velocity of releases (i.e., the pace with which they can release new versions).

As Facebook is one of the main drivers in the professional developer scene surrounding CD and continuous experimentation, the company is also commonly the subject of related studies. Feitelson et al. [2013] describe practices Facebook adopted to release on a daily basis. In a recent work, Savor et al. [2016] compared CD experiences at Facebook and OANDA and revealed that CD allows scaling in both the number of developers and code base sizes without decreasing productivity.

2.3.3 Continuous Experimentation

Experience Reports. There are also a multitude of academic publications discussing how key industrial players conduct continuous experiments. Tang et al. [2015] give insights how Facebook manages multiple versions running in parallel (e.g., using A/B testing) with a sophisticated configuration-as-code approach. There are also experience reports of Microsoft [Kohavi et al., 2013] and Google [Tang et al., 2010] on how they conduct experiments at a large scale. These works frame this research as a data science rather than a software or release engineering topic. In contrast, Kevic et al. [2017] investigated experimentation at Microsoft from a software engineering perspective. Using Bing as a case study, they investigated the complexity of the experimentation process and results show that code changes for experiments are four times larger than other code changes. Similarly, Fabijan et al. [2017] investigated the evolution of experimentation at Microsoft and presented a model detailing technical, organizational, and business evolution to provide a guidance towards data-driven experimentation.

Process and Design. Fagerholm et al. [2017] investigated the preconditions for setting up an experimentation system and characterized software instrumentation to collect, analyse, and store data as one of the challenges for experimentation. Bakshy and Frachtenberg [2015] provide guidelines for correctly designing and analyzing benchmark experiments. Bakshy et al. [2014] proposed a language for describing online field experiments, including A/B testing, at Facebook. Kohavi et al. [2009] provided a practical guide for conducting experiments. Tarvo et al. [2015] built a tool for automated canary testing incorporating the automated collection and analysis of metrics using statistics. Tamburrelli and Margara [2014] rephrase A/B testing as a search-based software engineering problem targeting automation by relying on aspect-oriented programming and genetic algorithms.

Implementation Techniques and Tooling. Rahman et al. [2016] analyzed the usage and evolution of feature toggles in 39 releases of Google Chrome and discussed their strengths and drawbacks. Recently, Veeraraghavan et al. [2016] described how Facebook uses a tool called *Kraken* to control (i.e. route) live user traffic on various levels (i.e., data center, server) to identify and resolve bottlenecks across their application ecosystem. Our own tooling, *Bifrost* [Schermann et al.,

2016b], supports the specification of experiments in a domain-specific language and uses runtime traffic routing for redirecting user requests to the right service versions.

2.3.4 Open Issues

Despite this significant body of work, we observe some relevant gaps. Primarily, the existing body of research uses case study research based on one, or very few, companies. In our work, we conduct a mixed-method study based on a larger sample size. Further, we focus on the software developer's or release engineer's point of view, rather than the perspective of managers, product owners, or data scientists. Lindgren and Münch [2016] recently did a step into a similar direction, focusing on a manager's perspective. They looked at the state of experimentation in 10 Finnish IT companies and came to the conclusion that it is not yet mature, as experimentation is rarely systematic and continuous. This is similar to what we have learned from some of our interview participants. However, we were also able to recruit several companies across multiple countries which make heavy use of experimentation. Other notable recent related research has been done by Shahin et al. [2016]. Their work focuses on practitioner reports from multiple companies regarding architectural issues of continuously deploying software. This work, which has been conducted in parallel to our study, largely comes to similar conclusions as we do regarding the importance of architecture for implemented experiments.

2.4 Research Methodology

We conducted a mixed-method study [Shull et al., 2007] consisting of two rounds of semi-structured, qualitative interviews combined with a quantitative survey. Figure 2.2 provides an overview of the research methodology. All interview materials and survey questions are part of the paper's online appendix[1]. Further details on the design and execution of our study complementing the information

[1] `http://www.ifi.uzh.ch/en/seal/people/schermann/projects/expstudy.html`

presented here can be found in our case study protocol [Brereton et al., 2008; Runeson et al., 2012] in the paper appendix. Prior to conducting the initial round of qualitative interviews, we performed a pre-study to identify practices associated with continuous experimentation.

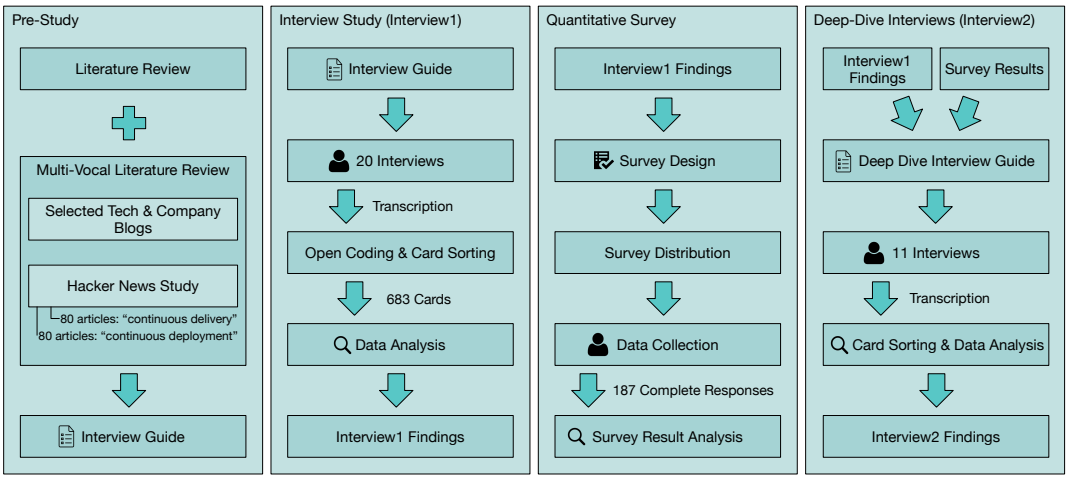

Figure 2.2: Overview of research methodology consisting of four steps.

2.4.1 Pre-Study

Protocol. The goal of the pre-study was to serve as a basis for formulating questions for the qualitative part of our study. As a starting point, we studied Rahman et al. [2015], Feitelson et al. [2013], Humble and Farley [2010], and the ThoughtWorks and Forrester Consulting [2013] report, which we considered standard CD literature at the time we conducted our pre-study (the mapping study by Rodríguez et al. [2016], which we also consider seminal for the field, was not yet available). In addition we studied multi-vocal literature [Garousi et al., 2016], i.e., unpublished or non-peer-reviewed sources of information usually produced by organizations or practitioners. This included studying tech blogs of industrial leaders such as Facebook [2016], Etsy [2016], Twitter [2016], Google [2016], and Netflix [2016]. These companies are known for conducting

experiments and using highly automated release processes, hence we used their blog posts to supplement the studied academic resources. To avoid potential bias introduced by our selection of blogs and inspired by Barik et al. [2015], we then used Hacker News[2] as an additional tool to identify further popular web resources. Articles were found using *hn.algolia.com*, a keyword-based Hacker News search engine. We searched for articles containing the keywords “continuous delivery" and “continuous deployment", which were posted between Jan 1 2011 and Nov 1 2015, and sorted them based on their popularity on Hacker News. For both keywords, we considered the first 80 articles. Our primary focus was on articles containing mainly experience reports, i.e., how companies make use of CD or continuous experimentation in the trenches. We removed those with dead links and those that mainly advertised specific tools. We ended up with 17 (continuous delivery) and 25 (continuous deployment) matching articles. We analyzed the articles based on the usage of CD and experimentation practices, compared them to the findings derived from literature, and created an interview guide divided into five themes: the release process in general, roles and responsibilities, quality assurance, issue handling, and release and experiment evaluation. A full list of articles, the detailed search criteria, and the resulting interview guide can be found in our online appendix.

Threats to Validity. The interview guide and the selection of questions for the qualitative phases of the study might have lead participants to answer towards our possibly biased notion of CD and experimentation (i.e., researcher bias). We mitigated this threat by building a foundation of understanding on the topic that is based on both previous academic work and online articles of well-known industry representatives. Potential bias introduced by our selection of these representatives is mitigated by including further experience reports and articles gathered via a keyword-based search on Hacker News. However, identifying whether a Hacker News article is suitable (i.e., whether it is an experience report or only advertising a service or tooling) introduced a further potential bias that we mitigated by having the authors discuss the relevancy of each article. Another limitation regarding the suitability and validity of the interview questions as a

[2] `https://news.ycombinator.com/`

result of the pre-study is that the first author designed all the questions. However, they were rigorously reviewed and verified by the other authors. In addition, some of the questions got improved based on participant feedback during the study.

2.4.2 Qualitative Interview Study (Interview$_1$)

Protocol. Based on the interview guide generated in the pre-study, we then conducted a first round of interviews. We fostered an exploratory character via a semi-structured interview process. All interviews included the mentioned five themes and discussion of each theme started off with an open question. Except for the first theme, topics were not covered in any particular order, but instead followed the natural flow of the interview. In total, the interview guide for this phase consisted of 52 questions. However, we did not ask every single question to each participant. The questions we asked rather depended on the flow of the interview and thus whether certain follow-up questions for the five themes were promising. Both, open and follow-up questions, can be found in our online appendix. The interviews were conducted by the first, the second, and the fourth author, either on-site in the areas of Zurich and Vienna, or remotely via Skype. All interviews where held in English or German, ranged between 35 and 60 minutes, and were recorded with the interviewee's approval.

Participants. We recruited interviewees from industry partners and our own personal networks, and increased our data set using snowball sampling [Atkinson and Flint, 2001], i.e., by asking existing interviewees to put us in contact with further potential interview partners that they are aware of. In total, we conducted 20 interviews in this phase, with developers or release engineers (P1 to P20, one female) from companies across multiple domains and sizes (see Table 2.1 and Figure 2.3). These companies range in size from single-person startups to global enterprises with more than 100,000 employees, and are located in Austria, Germany, Switzerland, Ireland, Ukraine, and the US. To ensure a broad understanding of the impact of CD and continuous experimentation on software development, we interviewed practitioners with different levels of seniority (average 9 years, standard deviation 5 years) and different project roles.

However, we required that all participants have insights into (technical) details on their company's or project's release process. We primarily selected companies developing Web-based applications, as our pre-study has shown that this is the application model most amenable for continuous experimentation. However, in spirit with the exploratory nature of our study, we also included other application types when companies mentioned their use of CD or continuous experimentation. Although participants *P9*, *P10*, and *P11* are employed by the same company, their teams work on different products utilizing different technology stacks and release processes. Due to the nature of the interviews, some of the questions target personal opinions, while others target the process, team, or even company level. Consequently, when discussing and reporting the results we sometimes refer to the participant's companies or team.

Analysis. The recorded interviews were transcribed by the first two authors. We coded the interviews on sentence level without any a priori codes or categories. The first three authors then analyzed the qualitative data using open card sorting [Spencer, 2009] (683 cards in total), and categorized the participants' statements, resulting in the set of findings described in the following. All findings are supported by statements of multiple participants. All selected quotes of interviews held in German were translated to English.

Threats to Validity. For the objectives of this study it was important to recruit interview participants that are approximately evenly distributed between organizations of varying sizes, divergent domains, and backgrounds (years of experience and age of participants). Snowball sampling helped us to increase our sample size. However, a potential disadvantage of this strategy is that it may suffer from community bias, as the first participants are prone to impacting the overall sample. We addressed this threat by selecting study participants purposefully, focusing on practitioners and also reviewed their online profiles, especially for those participants which were suggested to us via snowballing. Further, a potential threat to our empirical findings is that our results are not generalizable beyond the subjects involved in the interviews. We mitigate this effect by employing a mixed-method study validating our interview findings in a more general context using a quantitative survey in the following step. Furthermore,

we rely on self-reported (as opposed to observed) behavior and practices (self-reporting bias). Hence, participants may have provided idealized data about the CD and experimentation maturity of their companies. Furthermore, it is possible that we introduced bias through the mis-interpretation or mis-translation of "raw" results (interview transcripts). To avoid observer bias, these results were analyzed and coded by three authors of the study.

ID	Company		Application		Interviewee			Team Size
					Role	Experience (in Years)		
	Type	Country	App. Type	App. Domain		Total	In Company	
P1	SME	AT	Web	Sports News & Streaming	DevOps Engineer	3	3	3–6
P2	SME	AT	Enterpr. SW	Document Composition	Software Engineer	4	4	3–5
P3	SME	CH	Web	Employee Management	Software Engineer	10	5	1–3
P4	SME	CH	Web	Telecommunication	Software Engineer	15	4	3–7
P5	SME	AT	Web	Online Retail	Software Architect	5	5	15–20
P6	SME	AT	Desktop	SharePoint	Software Engineer	4	4	2–7
P7	Corp.	UA	Web	Employee Management	Software Engineer	5	5	4–6
P8	SME	AT	Enterpr. SW	Insurance	Software Engineer	12	12	5–8
P9	SME	CH	Enterpr. SW	E-Government	Solution Architect	13	13	4–6
P10	SME	CH	Web	Mobile Payment	Solution Architect	16	6	60–70
P11	SME	CH	Web	Mobile Payment	Solution Architect	11	4	15-20
P12	Corp.	DE	Web	Cloud Provider	DevOps Engineer	1	1	9–11
P13	Startup	AT	Web	Online Code Quality Analysis	DevOps Engineer	16	1	1
P14	Corp.	IE	Web	Network Monitoring	Public Cloud Architect	10	1	6–8
P15	Corp.	US	Web	Cloud Provider	Program Manager	15	3	8–10
P16	SME	AT	Enterpr. SW	E-Government	Project Lead	15	9	3–7
P17	Startup	US	Web	Babysitter Platform	Software Engineer	4	2	6–8
P18	Startup	US	Web	Event Management	Director of Engineering	5	1	5–7
P19	SME	US	Web	E-Commerce Platform	Software Engineer	5	3	3–7
P20	SME	AT	Embedded SW	Automotive Software	Software Engineer	3	3	3–5
D1	SME	US	Web	CMS Provider	DevOps Engineer	10	1	3–5
D2	SME	DE	Web	Q&A Platform	Head of Development	10	3	4–7
D3	Startup	CH	Web	HR Software	Head of Development	10	7	4–5
D4	SME	DE	Web	Travel Reviews & Booking	Software Engineer	7	2	5–7
D5	SME	DE	Web	Travel Reviews & Booking	Software Engineer	8	2	4–6
D6	Corp.	CH	Web	Telecommunication	Team Lead	5	4	7–9
D7	Corp.	UK	Web	Scientific Publisher	Director of Engineering	9	3	3–12
D8	SME	CH	Web	Network Services	Team Lead	30	3	5–8
D9	Corp.	US	Web	Video Streaming	Head Release Engineering	19	3	5–9
D10	SME	CH	Web	Sustainability Solutions	DevOps Engineer	10	8	1–4
D11	Corp.	CH	Web	Telecommunication	Software Engineer	10	2	5–10

Table 2.1: Interview study participants of both rounds of interviews

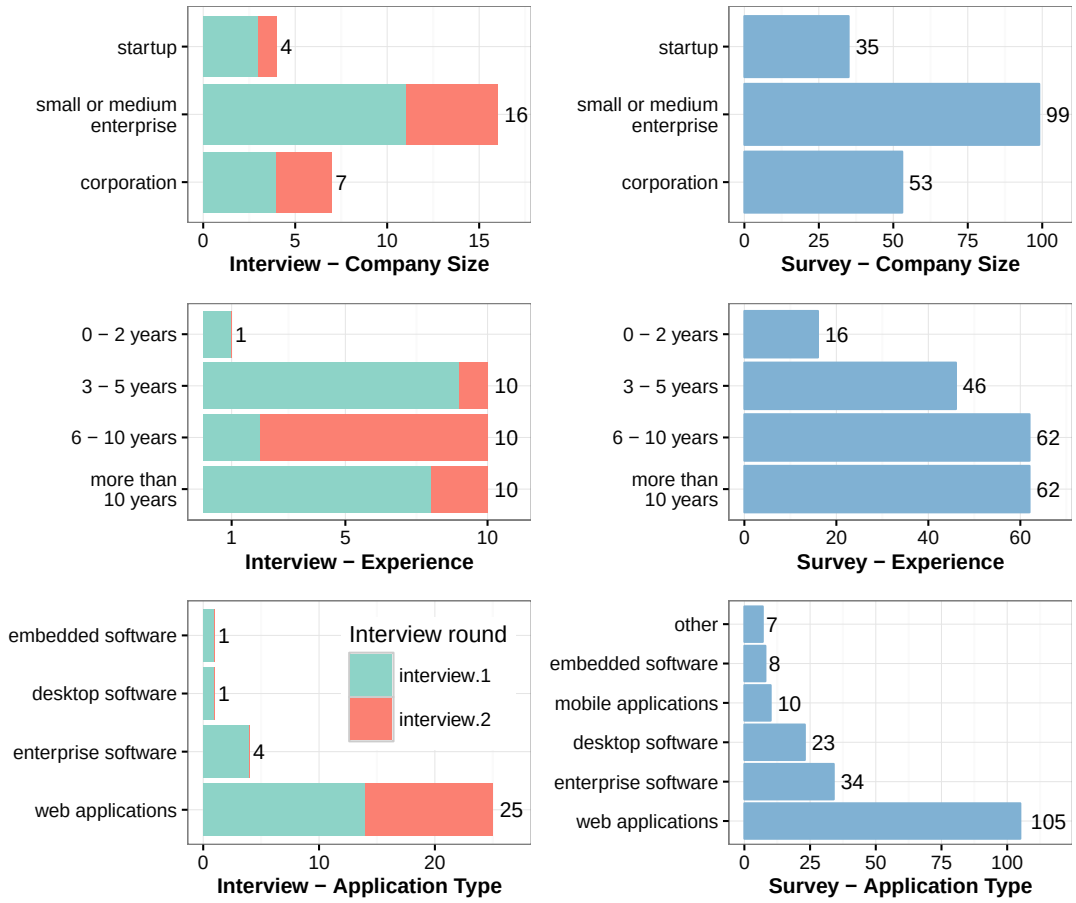

Figure 2.3: Demographics of interview study participants (left) and survey participants (right) subdivided into company sizes (top), experience (center), and application type (bottom).

2.4.3 Quantitative Survey

Protocol. To validate and substantiate the findings from our qualitative interviews on a larger sample size, we designed an anonymous Web-based survey consisting of, in total, 39 questions. Similar to the first round of interviews, we structured the survey into multiple themes: release process in general, software deployment, and issues in production. The survey mainly consisted of a combination of multiple-choice, single-choice, and Likert-scale questions. Although the

survey had its focus on quantitative aspects, we also included some free-form questions to gain further thoughts and opinions in a more qualitative manner. Depending on individual responses, we displayed different follow-up questions (i.e., branches in the survey) for the purpose of identifying underlying reasons (e.g., reasons for making use of canary releases, and reasons against). In total we had 7 branches (i.e., 7 mandatory questions) in our survey, thus the number of questions a participant had to answer varied. With this survey design we wanted to avoid presenting a participant with questions that do not make sense based on her previous answers.

Participants. We distributed the survey within our personal networks, social media, via two DevOps related newsletters[3,4], and via a German-speaking IT news portal[5]. As monetary incentives have been found to have a positive effect on participation rates [Smith et al., 2013], we offered the option to enter a raffle for two Amazon 50$ gift vouchers on survey completion. In total, we collected 187 complete responses (completion rate of 28% out of 667 responses). On average, it took the participants 12 minutes to fill out the survey. The survey was available online for three weeks in February 2016. Survey participants reported an average of 8 years of relevant experience in the software domain (standard deviation 4 years). Similar to the interviews, for some questions we were interested in the development and deployment process on the team or company level. Hence, we sometimes stick to the company level when discussing and reporting results. The resulting participant demographics for the survey is summarized on the right part of Figure 2.3.

Analysis. We analyzed the distributions of responses to Likert-scale, multiple-choice, and single-choice questions. In particular, we have correlated survey responses with the application model (Web-based or other) and the company size, as these two factors have emerged as important factors of influence in the interviews. Further, we coded the answers to open questions in the same style as for the interviews.

[3] http://www.devopsweekly.com/
[4] http://sreweekly.com/
[5] http://heise.de

Threats to Validity. We advertised our survey over various social media channels to attract a high number of respondents. However, participation in online surveys is necessarily voluntary. Hence, it is likely that the survey has attracted a respondent demography with substantial interest and familiarity with CD and experimentation practices (self-selection bias). Furthermore and similar to our interviews, participants may have provided idealized data about their companies' states on CD and experimentation (self-reporting bias). We piloted the survey with a small initial set of practitioners and gathered feedback to improve the survey before distributing it to a larger community and to avoid potential sources of ambiguity. Similar to our interview transcripts, there is the possibility that we introduced bias through mis-interpreting or mis-translating "raw" results gathered from the free-form questions in the survey.

2.4.4 Qualitative Deep-Dive Interviews (Interview$_2$)

Protocol. When revisiting our interview and survey findings, we identified the following topics to be of particular interest: (1) experiment design (e.g., metrics, hypotheses, duration), (2) implementation techniques for experiments, and (3) experiment result interpretation. In order to get more profound insights, we defined a set of 32 more detailed questions and conducted a second round of structured interviews. We followed the same protocol as in the initial interview round. Interviews lasted between 20 and 30 minutes and were again recorded with the interviewee's approval. Note that, we did not ask every single question to each participant, as this would have exceeded the targeted time frame. The questions we asked depended on the flow of the interview and thus the different techniques applied by the participant's company or team.

Participants. We again recruited participants from our personal networks and through snowball sampling. In total, we conducted 11 additional interviews with developers or release engineers (D1 to D11) from 9 different companies in various domains located in Germany, Switzerland, the United Kingdom, and the US (see Table 2.1 and Figure 2.3). Participants *D4* and *D5*, and participants *D6* and *D11* are employed by the same companies. However, as in the first interview phase, all participants work on different teams, and participants *D6*

and *D11* also work on different products. On average, participants of the second round of interviews had 12 years experience (standard deviation 7 years). All of the selected companies for the second round of interviews develop Web-based applications.

Analysis. The recordings of the second round of qualitative interviews were transcribed by the first author. The first three authors again used open coding to categorize the participants' statements and to gather more profound insights into continuous experimentation.

Threats to Validity. For the second round of qualitative interviews we are subject to the same threats to validity as in the first round of interviews that the reader should keep in mind when interpreting our results. We again recruited interview participants that are evenly distributed between organizations of varying sizes, divergent domains, and backgrounds. However, in this phase of interviews we focused solely on companies developing Web-based applications. As we were especially interested in continuous experimentation, we provided potential interview candidates with a brief outline of the goals of our study. While this allowed us to filter for participants that could provide us with useful information, this also introduced a potential threat that they shared information based on what they thought we wanted to know (i.e., hypothesis guessing), or withheld information or opinions that they thought would be unpopular (i.e., evaluation apprehension) [Wohlin et al., 2000]. We mitigated this threat by assuring that both their answers and company affiliation would be anonymized.

2.5 Practices for Continuous Experimentation

In this section, we cover best practices that facilitate continuous experimentation which emerged from our study. We start with technical practices (e.g., automation, architectural considerations) and move on to more organizational and cultural topics (e.g., awareness, developer on call).

2.5.1 Technical Practices

Automation and CI. To enable continuous experimentation, companies need to invest in deployment automation. A common implementation in CD projects are deployment pipelines [Bass et al., 2015; Humble and Farley, 2010]. Such pipelines consist of multiple defined phases a change has to pass until it reaches the production environment. The intrinsic goal behind investments in CD is to increase velocity, i.e., the time needed to pass all the quality gates and approval steps until a change reaches the production environment, while at the same time ensuring that the quality of the resulting product stays high [Schermann et al., 2016a]. Recently, there has been a multitude of research works on the challenges companies face on their way adopting CD, including technical and organizational [Chen, 2015; Leppanen et al., 2015; Olsson et al., 2012], as well as social challenges [Claps et al., 2015]. Our findings on obstacles regarding deployment automation are in line with existing research, including companies' internal policies (e.g., testing guidelines that are too strict in case of *P4*), or customers which do not appreciate higher release frequencies (e.g., *P9*).

Concerning continuous integration (CI), an often-cited prerequisite for CD and continuous experimentation [Humble and Farley, 2010], all but one company have embraced CI. However, CI has been widely covered by recent research. Hence, we omit a more detailed discussion on this topic and refer the reader to existing work (e.g., [Hilton et al., 2016; Vasilescu et al., 2015]) covered in Section 2.3.

Architectural Concerns of Continuous Experimentation. A suitable software architecture has been shown to be essential for experimentation, as it influences both, a company's velocity and release frequency:

> "*It is difficult to release individual parts of the system as dependencies between new code and the system in the back are just too high*" - *P5*

To tackle this problem, *P5* mentioned that in his company they have started migrating from their monolithic application architecture to smaller, independently deployable services (i.e., microservices) [Mazlami et al., 2017; Newman, 2015;

Schermann et al., 2015]. A similar result has also recently been independently reported by Shahin et al. [2016]. More generally, we have observed this trend across all our interviewees who use experimentation extensively. All of the companies they work for either have migrated to, or started from scratch with, a microservices-based software architecture. Different parts or functionality of a system are usually developed at a different pace and in different teams, so it comes quite natural that companies favor this option of independently deploying certain parts of their system. Another benefit our interviewees (e.g., *D7*) mentioned is that functionality is implemented with the technology which fits best, and non-monolithic architectures reduce the aversion of experimenting with more recent technology stacks. However, migrating to or designing architectures with many loosely-coupled entities bears its own risks. Suboptimal design decisions (e.g., using a central database for all services) lead to painful releases involving costly coordination among multiple teams whenever database schema changes occur (e.g., *D6*). However, once monoliths are broken down into multiple services (e.g., 70 – 80 services for *D4's* company, hundreds of services in case of *D9*), identifying the root causes of production issues becomes more challenging:

> "*[Root cause analysis] is difficult, and that's one of the main problems we face and we still have to tackle. If there is a severe issue and something is not working, guesswork starts, everyone's asking about reasons and trying things out*" - *D4*

Many teams and services are involved in troubleshooting these distributed problems. Traces of failed requests need to be carefully analyzed, and multiple deployments and their changes and running experiments have to be considered. One approach to tackle this is by forming a separate, centralized team or task force supporting the decentralized service teams.

> "*[...] they will get all the services in that area on basically a Slack channel, and then relevant engineers will start looking at their services and it's like a war room.*" - *D9*

Implementation Techniques. We observed multiple options on how to technically implement continuous experimentation. There is no "one size fits all" solution, and many companies combine multiple implementation techniques.

Feature Toggles. The implementation technique for continuous experimentation that was named most frequently in our study are feature toggles [Hodgson, 2016; Rahman et al., 2016]. They are used for canary testing and for gradual rollouts (e.g., *D2*, *D9*), for hiding not yet finished features in production code (e.g., *D7*, *P20*), to bucket users into groups for A/B testing (e.g., *P19*), or for dark launching new functionality (e.g., *D9*). Interestingly, some of our interview participants associated feature toggles with permission mechanisms, e.g., for regulating user access to specific features (e.g., *P9*, *D6*). *D2* appreciate that properly managed and synchronized (e.g., using tools such as ZooKeeper[6]) feature toggles give them more control over their application ecosystem:

> "*We do [feature toggling] on backend and frontend services, and especially on our iOS and Android apps because of their restricted (app store) release cycles. You want to be sure that if something is wrong, you can turn it off immediately across all frontends.*" - *D2*

As also reported by Rahman et al. [2016], our interviewees mentioned technical debt [Kruchten et al., 2012] and the additional level of complexity feature toggles add to systems (e.g., *P13*, *D6*) as major drawbacks. As Hodgson [2016] stated, feature toggles are easy to use, but they come with a maintenance cost. *D2* mentioned that they reached a point where continuously maintaining and testing 150 feature toggles became infeasible due to state explosion. Issues appeared when someone inadvertently flipped a flag and reactivated dead code. As a consequence, they drastically reduced and limited the number of feature toggles that are allowed to be active at the same time.

> "*I'm not using feature toggles and I don't intend to do so [...] Configuration leads to complexity, and every time you add complexity, you end up with additional complexity when you have to remove it at some point.*" - *P13*

[6] `https://zookeeper.apache.org/`

Runtime Traffic Routing. Besides feature toggles, another common implementation technique is runtime traffic routing (e.g., *D2*, *D5*). Depending on request header information (e.g., set cookies, device information), user requests are routed to selected backend instances, and, consequently, to specific versions of the software.

> "*When we could not make it with feature toggles (about 20 – 30% of the cases), we had to think about alternatives. In case of AdSense and Optimizely we set a cookie such that a user always gets the same version.*" - *D2*

A special type of traffic routing that is commonly used among our interview participants' companies are blue/ green deployments [Bass et al., 2015]. They use blue/green deployments mainly for canary testing followed by gradual rollouts. Once the first instance of new version works as expected, the remaining old instances are replaced in a stepwise manner, until a full rollout is reached.

Early Access. A final, relatively conservative, variation of continuous experimentation among our participants is providing specific users or user groups early access to binaries (e.g., *P8*). The main advantage of this model is, unlike for instance traffic routing, that it is not specific to Web-based applications. However, the downside is that the application provider has limited control over their experiments, and cannot, for instance, enforce the usage of the new version for specific users. Further, this experimentation scheme requires substantial manual and administrative effort.

Our interview findings are partially in line with our survey respondents (see Table 2.2). We use a color coding scheme throughout the tables of this paper in which darker cell background colors emphasize higher percentage values. Due to our focus on companies offering Web-based products in the qualitative parts of our study, we had only one company (*P8's* company) providing their software in form of binaries, as opposed to our survey participants with 29%. Regarding our survey participants, feature toggles are especially used by companies providing Web-based products (45%), while they are less frequently used for other application models (25%). While traffic routing is also frequently used for Web-based products (45%) among our survey participants, it is less important in

other application types (12%), in which pre-access to binaries is more common (47%).

	all n=70	Web n=38	other n=32	start. n=8	SME n=43	corp. n=19
other	6%	8%	3%	12%	5%	5%
permissions	17%	18%	16%	38%	16%	11%
dont' know	20%	13%	28%	12%	21%	21%
binaries	29%	13%	47%	12%	33%	26%
traffic routing	30%	45%	12%	38%	23%	42%
feature toggles	36%	45%	25%	50%	35%	32%

Table 2.2: Implementation techniques in use for continuous experimentation (multiple-choice).

Monitoring. An effect of highly automated pipelines is that not only new features reach production faster, but so do bugs. While delivery pipelines typically consist of a number of automated or manual quality checks, bugs are bound to slip through on occasion. This changes the way how companies have to deal with issues:

> "*I think the faster you move, the more tolerant you have to be about small things going wrong, but the slower you move, the more tolerant you have to be with large change sets that can be unpredictable.*" - *P18*

Highly automated pipelines allow companies to fix those small issues fast. Monitoring is a prerequisite for keeping developers aware of events in production environments. With continuous experimentation, the importance of monitoring applications even increases. Monitoring is not only used to determine if everything runs as expected (i.e., through health checks), but also to support rollout decisions (e.g., increase traffic assigned to a canary release) and decide about the continuation of ongoing experiments and the outcome of completed experiments (e.g., determining the outcome of an A/B test).

> "*The decision whether to continue rolling out is based on monitoring data. We look at log files, has something happened, did we get any customer feedback, if there is nothing for a couple of days, then we move on.*" - *P16*

Interview participants mentioned that they do not only rely on monitoring data to identify runtime issues, but also take customer feedback, for instance provided via bug reports, into account. This was also supported by our survey results. Customer feedback (85%) and active monitoring (76%) are both widely used among survey respondents (see Table 2.3). For Web-based applications, monitoring and customer feedback are in balance, while for other application types, customer feedback (90%) is dominant (67% monitoring). This is not surprising, as monitoring Web-based applications is technically easier than for other application models, and supported by existing Application Performance Monitoring (APM) tools, such as New Relic [Ahmed et al., 2016].

	all n=187	**Web** n=105	**other** n=82	**start.** n=35	**SME** n=99	**corp.** n=53
don't know + other	4%	2%	6%	3%	5%	2%
monitoring	76%	83%	67%	89%	72%	75%
customer feedback	85%	81%	90%	80%	88%	83%

Table 2.3: How issues are usually detected (multiple-choice).

2.5.2 Organizational and Cultural Practices

Awareness. Awareness refers to activities that foster transparency of the development and experimentation process for every stakeholder (e.g., developers, testers, operations). Similarly to monitoring, awareness is becoming even more important once continuous experiments are conducted. Multiple deployments and experiments conducted at the same time can negatively influence data collection and statistically robust analysis, i.e., correctly identifying and dealing with the noise induced by concurrent experiments. Consequently, it is important that developers, release engineers, and other stakeholders stay informed. We distinguish

between awareness throughout the development process, and during experimentation. The former typically covers tooling that tracks status or progress of features through tasks or tickets (e.g., Pivotal tracker). The latter involves various ways of informing other teams about experiments being conducted, e.g., internal wiki or blog posts (*D1*), e-mail notifications (*D9*), or meetings of product owners and team leads (*D2*). Combined solutions involve online dashboards, or public screens in the office, which display information such as build status, test results, or production performance metrics. Another way to promote awareness and transparency is through signals sent in the form of asynchronous communication tools that are integrated with the team collaboration chat tools, such as Slack or HipChat [Lin et al., 2016].

Developer on Call. Interviewees agree that the notion of developer on call, i.e., that a developer needs to be available to provide operational support after a release, has become a widely accepted practice in their organization. This was not only the case for companies following a service-based architecture, where being responsible as a team for your own services comes naturally, but also for other companies we interviewed. In case of issues, developers know best about their changes and can help operations to identify the problem faster and contribute to the decision about subsequent actions. Additionally, *P16* also specifically mentions a learning effect for developers:

> "*Developers need to feel the pain they cause for customers. The closer they are to operations the better, because of the massive learning effect.*" - *P16*

This practice is strongly related to DevOps and emphasizes a shift in culture that is currently taking place. Traditional borders between development, quality assurance, and operations seem to vanish progressively. This addition of responsibility could lead developers to writing and testing their code more thoroughly, as some participants indicated:

> "*If you don't have enough tests and you deploy bad code it will fire back because you would be on call and you have to support it*" - *P14*

Some participants (e.g., *P7*) mention that their companies avoid the additional burden of keeping developers on call on weekends by releasing only during office hours. However, for many companies and domains, deployment weekends are a business necessity (e.g., *P9*). Others follow a more pragmatic process with a clear handover of responsibility. For instance, at *D5's* company, developers provide a manual containing step by step descriptions for operations on how to act in certain circumstances (e.g., rollbacks, flipping a feature toggle to turn off the experiment).

Our survey confirmed these findings (see Table 2.4). The majority of survey respondents stated that developers never hand off their responsibility for a change. When comparing company sizes, developers are on call particularly at startups (74%), but even in larger corporations this concept is applied frequently (45%). While in SMEs and corporations (23%) developers hand off their responsibility directly after development, this is almost never the case for startups (3%).

	all n=187	**Web** n=105	**other** n=82	**start.** n=35	**SME** n=99	**corp.** n=53
don't know + other	4%	2%	5%	3%	1%	8%
preproduction	9%	10%	9%	9%	8%	11%
staging	12%	15%	9%	11%	12%	13%
development	19%	12%	28%	3%	23%	23%
never	56%	61%	50%	74%	56%	45%

Table 2.4: Phase in the release process after which developers typically hand off responsibility for their code (single-choice).

Decentralized Teams and Consultants. Many interview participants are not only supported by central teams providing infrastructure (e.g., deployment pipelines, containers with pre-configured monitoring) and tooling, but also by a range of consulting teams. In companies that adopt microservices, teams developing functionality are autonomous in most of their decisions, including experimentation. However, not all teams are staffed with experts in all relevant fields to either conduct or interpret experiments (e.g., data scientists, DevOps engineers). These teams can request support from centralized teams, e.g., for

identifying the right set of metrics and thresholds to assess a service's health state (e.g., *D9*, *D1*). We further observed that tooling and infrastructure provided by a centralized team increase technology homogeneity, since they not only provide, but also maintain, standard tools:

> "*[...] you are allowed whatever tool you want. The interesting thing is, [...] teams are not required to use [tool name] if they don't want to, but everyone uses it*" - *D9*

Teams using their own technology stacks are required to maintain them, leading to additional effort. Further, the service team is held responsible when their non-standard tools fail or lead to other issues.

2.6 Conducting Experiments

After covering common practices, this section focuses on how companies actually conduct continuous experiments. A central aspect that emerged from our study is that there are fundamentally two classes of experiments, namely experiments conducted to identify and mitigate the impact of software regressions, such as functional bugs that evaded detection in the delivery pipeline, performance regressions, or scalability issues (*regression-driven experiments*) and experiments conducted to evaluate different software design or implementation decisions from a business perspective (*business-driven experiments*). While superficially similar on a technical level, different concrete practices are typically used to implement those classes of experiments. Business-driven experiments are primarily conducted using A/B testing. For regression-driven experiments, multiple techniques are in use, including canary releases, dark launches, and gradual rollouts. We summarize the main characteristics, differences, and commonalities of these classes of experiments in Table 2.5.

2.6.1 Regression-Driven Experiments

This variant is about mitigating technical risks and verifying the correct functioning of a new version or feature. Regression-driven experiments are used to detect

	Regression-Driven Experiments	Business-Driven Experiments
Main Goals	Mitigation of technical problems (e.g., related to bugs or performance regressions), conducting health checks, testing scalability on production workload	Evaluation from a business perspective of new features or different implementation decisions (do customers appreciate the change, is it in line with monetary incentives and company goals?)
Common Practices	Canary releases, dark launches, gradual rollouts	A/B testing
Used Metrics	Typically multiple application and infrastructure level metrics (e.g., response time) in combination with simple-to-measure business metrics	Primarily business metrics, sometimes combined with small selection of application metrics
Data Interpretation	Often intuitive and based on experience, less process driven (do metrics "seem higher than before"?)	More statistically rigorous hypothesis testing based on carefully selected metrics
Experiment Duration	Minutes to multiple days	Often in the order of weeks (see also Kevic et al. [2017])
Selection of Target Users	Often small scoped (e.g., small percentage of users, user groups, regions), sometimes gradually increased until full rollout	Two or more groups (percentage of user base, user groups, regions) of same size, constant size during experiment
Responsibility	Siloization, single team or developers	Multiple teams and services involved, requires coordination, awareness, and commitment across team borders
Impl. Techniques	Feature toggles, dynamic traffic routing, distribution of different variants in form of binaries	
Main Obstacles	Architecture, limited number of users, missing business value or not worth investments, lack of expertise	

Table 2.5: Summary and comparison of regression-driven and business-driven experiments.

functional problems that slipped through unit or integration testing, performance regressions, or new features that do not scale to production workloads.

> "*Even though [a new feature is] tested in test, it's still the data combinatorics in production are so vastly different than we can simulate in test that in some cases we do find issues in production.*" - *D9*

Such production "health checks" are implemented in various ways and on differing scales. A commonly used practice among our interview participants are canary releases. Release engineers either make use of them for all changes, or, more commonly, use this practice for specific changes that are considered particularly critical. A typical use case is scalability testing in Web-based applications (e.g., *P4*, *D2*).

> "*[We use canary releases] especially in those cases when we have concerns how it would scale when all users get immediate access to this new feature.*" - *P4*

Our survey has shown that 63% of practitioners are not using any variant of regression-driven experimentation (Table 2.6). Consistent with our interview results, this flavor of experimentation – among those that actually make use of it – is not bound to companies developing Web-based applications. There is no significant difference in our survey responses in terms of its adoption between developers of Web-based applications and others. However, for developers using other application models, partial rollouts usually come in the form of simple pilot or early access phases. These are usually manually-administered with hand-picked friendly customers (e.g., companies of *P8*, *P9*, *D3*). This concept is similar to pre-release versions (e.g., alpha, beta, RC) sometimes used in desktop and enterprise software. Such early-access canaries are typically not systematically monitored and experiment outcomes are determined primarily by analyzing user feedback.

	all n=187	**Web** n=105	**other** n=82	**start.** n=35	**SME** n=99	**corp.** n=53
for all features	18%	15%	22%	6%	22%	19%
for some features	19%	21%	17%	17%	21%	17%
no experimentation	63%	64%	61%	77%	57%	64%

Table 2.6: Usage of regression-driven experimentation (single choice).

Dark or shadow launches, as pioneered by Facebook, are rarely used among our interviewees. Only *D9* conducts dark launches in a similar fashion as described

by Facebook, by implementing and controlling experiments using feature toggles. *D1* mentioned that they do not have the necessary scale for it, and *D2* does not see a pressing need. *D5*, however, occasionally conducts a simplified version of dark launches:

> "*We do have a procedure such that as long as a service [version] is not effectively enabled in production we push every feature branch to prod, thus we can ensure that it runs as we expect [...] including [real or generated] traffic would be the next logical step.*" - *D5*

Metrics. In case of canary releases, measured metrics consist of standard application (e.g., response times) and infrastructure (e.g., CPU utilization) metrics. This is consistent with our results from a previous study [Schermann et al., 2015]. Interviewees did not have strict rules on what to monitor, nor do they have access to clear thresholds or tests that help them assess whether specific monitoring data should be considered "healthy" for a given application. Instead, practitioners conduct health assessments iteratively and primarily based on intuition. If a metric value appears problematic (e.g., appears to be visually different in a dashboard), they take action based on informal past experience rather than well-defined processes and empirical data. This is consistent with our experiences in earlier studies [Cito et al., 2015a,b]. If formal thresholds are used, they are often based on historical metrics gathered from previous releases. Even though a minority in our study population, some interviewees (e.g., *D2*, *D4*, *D9*) also used *a priori* defined metrics and thresholds.

> "*On a low level basis, [...] [we] basically do an apples to apples comparison for about 2000 metrics, so every team is kinda free to pick their own. [...] they are looking for deviations [...] if you spin up version 2, it does a comparison and then you can basically say what the variance is allowed to be.*" - *D9*

Notifications are typically sent automatically if the data shows any (negative) deviations from the baseline version. *D5* mentioned that setting concrete thresholds is tricky and often leads to false alarms. Hence, they refrain from setting specific thresholds.

Responsibility. In microservices-based architectures, which many of our interview participants use extensively (*P10*, *P12*, *P14*, *P15*, *P19*, *D2*, *D4*, *D5*, *D7*, *D9*), regression-driven experiments are often characterized by "siloization". Teams responsible for a service decide when and how long to conduct experiments on this service. Moreover, it is typically the task of the team to interpret monitoring data collected during the experiment. However, not all teams have the necessary data science or domain knowledge to do so with confidence. Hence, centralized support teams are sometimes available that help identify metrics to look after, interpret collected data, and identify issues causing experiments to fail (e.g., *D1*, *D4*, *D9*). In other companies (e.g., *P4's* company), conducting experiments is a shared task between release engineers, team leads, and operations, which is outside the traditional microservice team structure.

Duration and User Selection. The duration of canary tests, dark launches, and gradual rollouts varies from few minutes (e.g., in case of *D7*, whose team conducts very short-term 5-minute health checks) to multiple days, but rarely takes longer than two weeks. The end of the spectrum includes those companies rolling out on a data center level (e.g., companies of *P12*, *D8*) or directly contact their customers for feedback (e.g., through early access phases in case of *D3* and *P8*). The amount of, and which, users are considered for an experiment depend on a new feature's complexity, i.e., the more critical a feature, the higher the risk, thus the smaller the scope of the experiment initially. User selection varies and involves random selection on user traffic level, specific user groups (e.g., role, device), or entire regions and countries. Some companies (e.g., of *P16* or *D1*) apply further risk mitigation strategies by following an "eat your own dog food" [Moskowitz, 2003] approach. That is, they are rolling out and testing new versions of their software internally first before rolling out to external customers.

2.6.2 Business-Driven Experiments

The primary purpose of business-driven experimentation, most commonly associated with A/B testing, is to evaluate the business value of specific features, implementation decisions, or products. Prerequisite for business-driven experimentation is that the versions under test are technically sound. In our study, the

central system-under-test for this type of experiments were user facing frontends. A special case was the company of *D5* that relied on A/B testing also for their migration from a monolithic to a microservices-based architecture. The company of *D7* sometimes conduct, as they called it, "fake A/B tests", in which they were interested in the demand for a certain feature without actually implementing it due to high costs and unknown demand. They integrated a mockup into the user interface and kept track of user interactions.

> "*We used it as our decision basis, in that mentioned case we implemented the feature because data have shown that it generates more downloads and thus more money*" - *D7*

23% of our survey respondents have adopted A/B testing. Interestingly, this practice is not only bound to companies developing Web-based applications, even though they still represent the majority with 63% of A/B test users in our survey. Consistent with our interviews, evaluating changes in the user interface is the most common use case (88%) in our survey, but backend features are also A/B tested by 44% of the respondents.

Metrics. Due to their higher strategic importance, decision making in business-driven experiments tends to be governed less by intuition and experience, and more by statistically sound data analysis. Companies more often start experiments with clearly defined hypotheses, deciding a priori about what to expect (i.e., metrics and deviations), which users to invite or select, and how long the experiment should take. Our interviewees often had a selection of domain-specific key performance indicators (KPIs) they looked at specifically throughout those experiments, such as conversion rates or sales figures:

> "*It was about evaluating KPIs, how did they perform in both groups, what did we expect. Prerequisite is that you have to ensure during development that you can measure those metrics later on.*" - *D2*

Responsibility. Business-driven experiments often involve more than a single team. For instance, frontend functionality leverages multiple backend services, thus coordination and commitment among all teams along the call path

is required. Teams need to make sure that multiple experiments, both regression- and business-driven, do not negatively influence each other. Some companies (e.g., of *D2*) only allow exactly one experiment being conducted for a single part of the application (e.g., frontend site), while others (e.g., of *D1*, *D9*) tackle this problem through long test durations and large sample sizes, and treat other experiments simply as noise:

> "*There is the ability to see if it affected it, but I don't think we necessarily pay too much attention. [...] overall, A/B tests run for a long time, I think they evaluate this as noise*" - *D9*

Experiment data interpretation requires substantial expertise in statistics and data science. Interpretation is either a shared task (e.g., *D1*), or carried out by single team members, often product owners of frontends (e.g., *D2*, *D4*).

Duration and User Selection. The exact duration of business-driven experimentation varied, but was typically in the area of 4 to 6 weeks for our interviewees. Experiment durations are dependent on getting enough data to allow for statistical significant conclusions and to deal with fluctuations:

> "*Feature performance varied on a daily basis, could be different on day three than on day four, that's why we take enough time to [collect data and] draw valid conclusions.*" - *D2*

Similar to regression-driven experiments, user selection strategies vary, and can include random sampling, specific roles or user groups, and regions or countries. Moreover, the concrete user selection strategy depends on the actual feature being tested, and may require coordination with marketing and product development (e.g., in case of *P17*) as well.

In terms of size of test and control groups, we identified different approaches. *D2's* company uses the same sizes each time for test and control groups to facilitate data interpretation. 1% of the user traffic is used as test group for the new feature, and 1% of the user traffic as control group. The remaining 98% get the same version as the control group without being tracked. *D1* mentioned

that it depends on the teams experience, some conduct 50:50 scale experiments, others start with 2% versus 98% of user traffic.

2.6.3 Obstacles of Continuous Experimentation

We now report on the main problems and obstacles to adopting continuous experimentation, both of the regression- and business-driven variety. For the 63% of respondents that are not actually using any variation of regression-driven experiments, the largest obstacle is a software architecture that does not easily support experimentation. This was particularly evident for SMEs and corporations, and for companies that develop Web-based products (64%, versus 48% for others). It is likely that this is because most Web-based products in these domains are still deployed as monolithic 3-tier applications. For startups, software architecture is slightly less of a concern. However, startups often do not have a sufficiently large customer base to warrant regression-driven experimentation. This is linked to a third, similar problem preventing the adoption of this type of experiments – some teams or companies simply do not see any business value in conducting them. Interestingly, lack of expertise was only seen as a minor barrier for adoption, given by 26% of respondents overall. A summary of the main reasons against adopting regression-driven experiments is shown in Table 2.7.

	all n=117	**Web** n=67	**other** n=50	**start.** n=27	**SME** n=56	**corp.** n=34
other	18%	1%	10%	7%	4%	6%
lack of expertise	26%	27%	24%	15%	34%	21%
no business sense	39%	39%	40%	41%	36%	44%
number customers	39%	46%	30%	56%	38%	29%
architecture	57%	64%	48%	44%	66%	53%

Table 2.7: Reasons against conducting regression-driven experiments (multiple-choice).

A summary of the main reasons against business-driven experiments as resulting from our survey is given in Table 2.8. Unsurprisingly, and similarly to the obstacles for regression-driven experiments, for those 77% of participants

that are not making use of A/B testing, the biggest challenge is a software architecture that does not support running and comparing two or more versions in parallel. Unsuitable software architectures are mainly a problem for SMEs and corporations, while for startups a small user base is seen as a major obstacle. This also was an issue that emerged from our interviews:

> "*We only have around 130 customers, it is actually easier to just talk to everybody.*"
> *- P18*

Once enough data points are collected to ensure statistical power, expertise is needed to analyze and draw valid conclusions. However, a lack of expertise was only mentioned by a minority of respondents (15%) as a problem. Interestingly, many companies report that they do not have the features for which it would be worth conducting A/B tests. A similar theme has also emerged in the interviews. The return on investment, both financial and time, of creating and/or setting up appropriate tooling would be just too low. This was mentioned by 33% of our survey participants. While limitations because of internal policies are minor factors for startups, for corporations this represents a strong barrier.

	all n=144	**Web** n=78	**other** n=66	**start.** n=25	**SME** n=74	**corp.** n=45
other	6%	4%	8%	4%	1%	13%
don't know	6%	5%	6%	4%	7%	4%
lack of knowledge	15%	19%	11%	12%	15%	18%
policy / domain	21%	14%	29%	12%	22%	24%
number of users	28%	32%	23%	44%	27%	20%
investments	33%	35%	30%	44%	31%	29%
architecture	50%	53%	47%	40%	59%	40%

Table 2.8: Reasons against conducting business-driven experiments (multiple-choice).

2.6.4 Summary

Having covered the two flavors of continuous experimentation that emerged from our study, we now want to summarize the usage of continuous experimentation practices among our interview participants (i.e., including both interview rounds Interview_1, and Interview_2). Table 2.9 provides an overview of the prevalence of microservices-based architectures, the usage of implementation techniques (i.e., feature toggles, traffic routing, and early access to binaries), whether developers are "on call", and finally, whether development teams are supported by decentralized teams and consultants. Besides those practices, Table 2.9 also depicts if and of which classes of continuous experimentation (i.e., regression-driven and business-driven) the company or team makes use of. For each participant in our interview studies, we provide a simple mapping whether the participant's team uses (turquois), does not use (white), or partially uses (color graded turquois) a respective practice or type of continuous experimentation. Partial usage means that the respective company or team does have concrete plans to use a practice or is currently in the process of migration (e.g., moving from a monolithic towards a microservices-based architecture).

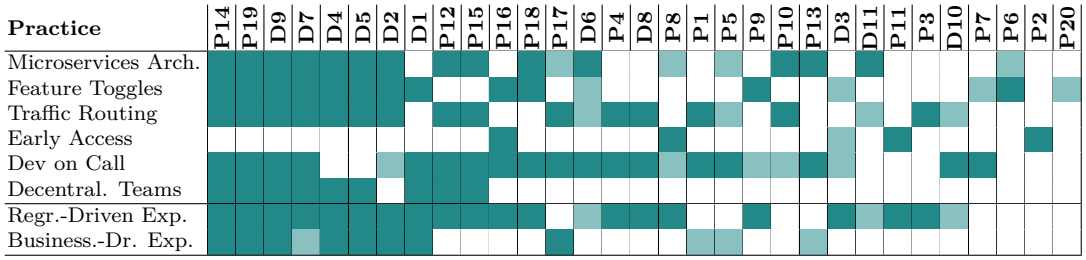

Table 2.9: Usage of continuous experimentation practices by our interview participants.

Developer on call is a widely accepted practice among our interview participants, while decentralized and consulting teams are especially common in larger organizations. Feature toggles and traffic routing are the typical implementation techniques for continuous experimentation. However, although being a niche practice, some of our interview participants prefer a more conservative

approach of providing certain users early access to the newest binaries. We also see that microservices-based architectures are strongly represented in those companies making extensive use of either regression-driven or business-driven continuous experimentation. Among our interview participants, regression-driven continuous experimentation is more common than business-driven continuous experimentation. However, four companies do have concrete plans for conducting business-driven continuous experimentation.

2.7 Implications

We now discuss the main implications of our study. We focus on the underlying problems and principles we have observed, and propose directions for future research.

Architectural support for experimentation. As discussed in Section 2.6, a (legacy) system architecture is a dividing barrier between companies that do and those that do not adopt continuous experimentation. Such an architecture makes advanced practices, such as canary releases or A/B testing, hard to implement. We have observed that applying feature toggles (see Section 2.5) to circumvent architectural limitations for implementing experimentation comes at the price of increased complexity, which negatively affects maintainability and code comprehension. Moreover, as reported by Rahman et al. [2016] they introduce technical debt. Microservices, or other architectural models that foster independently deployable components or services, are a promising enabling technology to ease experimentation, but the community is currently lacking formal research into the tradeoffs associated with such architectural styles. For instance, we have observed that practitioners currently lack means to decompose an application into microservices in the first place, or identify which microservice is causing a runtime issue along the call path. Further, more studies are needed to assess the suitability of microservices for various continuous experimentation practices.

Modeling of variability. Related to the previous implication, the results reported in Section 2.5 imply that practitioners currently struggle with the

complexity induced by feature toggles. Hence, it can be argued that more research is needed on better formalisms for modeling the software variability induced by feature toggles, as well as for their practical implementation without polluting the application's source code with release engineering functionality. There has been a multitude of research around variability, i.e., how software can be adjusted for different contexts (e.g., Galster et al. [2014], Capilla et al. [2013]). We suspect that concepts such aspect-oriented software development [Kiczales, 1996] and (dynamic) product line engineering [Hallsteinsen et al., 2008] could serve as useful abstractions in the domain of continuous experimentation. However, their usage did not emerge in our study even though these techniques have been available for years.

From intuition to principled decision making. In Section 2.6, we have observed that many release engineers are mostly going by intuition and previous experience when defining metrics and thresholds to evaluate the success of regression-driven experiments. Similarly, which features to conduct canary tests on, or which (fraction of) users to evaluate, is rarely based on a sound statistical or empirical basis. Hence, research should strive to identify, for various application types, the principal metrics that allow for evaluating the success of an experiment, and identify best practices on how to select changes that require experimentation. Further, robust statistical methods need to be devised that suggest how long to run at which scope (e.g., number of users) to achieve the required level of confidence. A main challenge for this line of research will be that release engineers cannot generally be expected to be trained data scientists. This is particularly true for smaller companies, for which release decision making needs to remain cost-efficient and statistically sound on a small sample size.

The many hats of developers. An underlying theme of our study results is that developers in those companies that use models such as DevOps, developer on call, microservices, or continuous experimentation, often need to juggle their core task of writing code against many other responsibilities, including operations support, release planning, and data analysis—often under considerable pressure to move fast [Rubin and Rinard, 2016]. While we have observed that some companies provide central support teams (e.g., through dedicated data science

consultants [Kim et al., 2016]), in many companies the teams need to acquire the necessary expertise to handle these new job aspects themselves. However, not only the software developer's role is subject to change in the context of CD and continuous experimentation, but also the software architect's. As reported by Hohpe et al. [2016], software architecture has also become broader and more complex, requiring practitioners to steadily keep informed about new technology such as microservices-based architectures. Designing complex systems is more than leveraging object-oriented design skills, it involves leading, mentoring, and conveying complex concepts in approachable terms. Follow-up studies will be required that address these changes in the job profile of software developers and architects.

2.8 Conclusions

We report on an exploratory, yet systematic, empirical study on the practices of continuous experimentation. Continuous experimentation guides development activities based on data collected on a subset of online users on a new experimental version of the software. As continuous experimentation is especially amenable for Web-based applications, we primarily selected developers or release engineers from companies developing Web-based applications for the qualitative phases of the study. The insights provided by our study help to understand the state of practice in this field, how companies use experimentation and which challenges they face when adopting it. First, many companies practice it as an experience-driven "art" with little empirical or formal basis. Our study suggests that foundational support is needed for moving towards principled approaches for release decision making. Second, small and independently deployable services (e.g., microservice architectures) have emerged as a enabling technology, named by all our interviewees who heavily use experimentation techniques. However, we found that guidelines are required on how to decompose (monolithic) applications and migrate to such microservices-like architectural styles. And third, the advent of experimentation and continuous deployment processes has led to a shift in responsibilities. Developer on call policies have become widely accepted, in which

developers are not only responsible for the code they write, but also decide in collaboration with their team which experiments to conduct and which metrics to consider for evaluation.

Once a more principled approach for release decision making is established, we envision this to lead to well-defined, structured continuous experimentation processes implemented in code (i.e., Experimentation-as-Code), analogously to the already established concept of Infrastructure-as-Code [Hummer et al., 2013]. Such Experimentation-as-Code scripts can be structured into multiple phases with clearly specified gateways and repair actions. This will not only provide means for further automation, but also facilitate the documentation, transparency, and even formal verification of experimentation processes. We have already proposed initial steps into this direction as part of our proof-of-concept system Bifrost [Schermann et al., 2016b].

2.9 Acknowledgments

The authors would like to thank all interview and survey participants. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610802 (CloudWave), and from the Swiss National Science Foundation (SNSF) under project name "Whiteboard" (SNSF Project no. 149450).

2.A Appendix - Case Study Protocol

For planning, conducting, and reporting our study, we followed the case study protocol proposed by Brereton et al. [2008]. We further considered the guidelines reported by Runeson et al. [2012]. The following sections provide details on study design, case selection, data collection, and analysis.

2.A.1 Background

The goal of our study was to identify (1) the principles and practices that enable and hinder organizations to leverage continuous experimentation, and (2) how companies use experimentation and how their techniques differ. In Section 2.3, we provide details on related work in the area of release engineering and list open issues we tried to address within the context of our study. Prior to starting the data collection process of our study, we conducted a literature review (i.e., our pre-study) to identify typical practices associated with continuous experimentation and derive a questionnaire for the first round of qualitative interviews (see Section 2.4.1 for details).

2.A.2 Design

Our study was designed as an embedded, multiple-case study. We followed a multiple-case design as we are interested in the state of practice in continuous experimentation. Rather than limiting our data collection to a single case study company, we aimed for a more comprehensive view on the field. We interviewed 31 developers or release engineers from 27 companies. The object of study was the release process of the participating companies. Depending on the size and the domain of a company it could be the case that multiple (different) release processes are in place (e.g., for different products, or projects). Within the context of our study, we focused on those processes our participants are associated with, i.e., the product or project they are working on. In three cases during the qualitative phases of the study we collected data from within the same company. For these cases, we ensured that data on projects or products with different release processes were collected. We chose an embedded study design since we are not only integrating data from multiple cases, but also analyze multiple embedded components from each case, i.e., multiple units of analysis. We further position our study as exploratory [Runeson et al., 2012], as it sought to generate new insights, and we adopted a "soft" case approach according to Braa and Vidgen [1999] as our research outcome was about gaining understanding. We complemented our qualitative data by conducting a quantitative online survey.

An important step in our study was the literature review (i.e., pre-study) to get a notion of the practices associated with continuous experimentation and to serve as a basis for developing a (first) questionnaire. Section 2.4.1 provides details on the pre-study, including the considered related research and multi-vocal literature, and the search criteria used. The literature review was not systematic (SLR). Rather, given the exploratory character of our study, we sought to identify a set of key concerns or themes (i.e., forming a theoretical framework) that are important considerations when reviewing the state of practice in continuous experimentation. These key concerns further establish the boundaries of our study and directly link to the units of analysis (i.e., release process, roles and responsibilities, quality assurance process, issue handling process, experimentation process, experimentation design, experimentation implementation techniques, and experiment result interpretation) and consequently, support us in answering our research questions introduced in Section 2.1.

Interview Study 1 - Key Concerns

In the following, we will briefly describe the key concerns and themes (i.e., units of analysis) covered in the first qualitative phase of our study.

Release process in general. The goal is to analyze the single phases (e.g., building, testing, deploying) a (code) change has to pass through once a developer pushes the change to the version control system triggering the release process. This allows us to get a first overview of the release process, whether development, quality assurance, and operations tasks are tightly connected or strictly separated within a company, and how automated the entire process is.

Roles and responsibilities. This concern sheds light on the various stakeholders involved in the software release process. Questions covered within this theme involve, amongst others, who decides to ship a certain feature, who is responsible for problems that might appear, are developers required to stay on call, and how aware are the various stakeholders about ongoing and upcoming releases and experiments.

Quality assurance. Within this theme we are interested in details how a company ensures software quality. This involves analyzing whether quality

assurance is separated into multiple stages within a so-called deployment pipeline, whether manual approving is necessary in between, and whether builds happen exactly once throughout the release process.

Issue handling. This concern deals with the process of handling issues, whether problems detected in the production environment are treated different than other issues and how those issues are typically detected (e.g., monitoring measures in place), by whom, and how long does it take to fix them.

Release and experiment evaluation. This concern should provide us insights into the experimentation process of a company or project. It unveils how companies keep track of experiments, whether there are strictly defined processes for testing new features on a small fraction of the user base, and how do the various stakeholders involved interact with each other.

Interview Study 2 - Key Concerns

In order to get more profound insights into the experimentation processes, we divided the release and experiment evaluation concern into three more detailed key concerns for the second round of interviews that are briefly sketched in the following.

Experiment design. This key concern covers the topic of how companies plan and design experiments. It helps us to determine whether experimentation follows a strict process (e.g., defining hypotheses, pre-selected set of metrics to monitor) or is more driven by the developer's gut-feeling and who is typically involved and responsible when designing and planning experiments.

Implementation techniques. The theme of implementation techniques covers the technical aspect of continuous experimentation. It sheds light on the various techniques (e.g., feature toggles, traffic routing) used for different types of experimentation (e.g., canary releases, A/B testing) and how these are combined.

Experiment result interpretation. Experimentation is all about data collection and data interpretation. We are interested in how companies interpret the collected data, in which intervals, and who is responsible for it.

2.A.3 Case Selection

Ideally, the cases (i.e., companies or projects in our study) should be selected intentionally and the units of analysis should have variation in their properties such that the application of data analysis methods reveal new insights. In our study, the recruited companies range in size from single-person startups to global enterprises. For the qualitative phases of our study, we selected companies or projects across multiple different domains (see Table 2.1). We primarily selected companies or projects developing Web-based applications, as our pre-study revealed that this is the application model which is most amenable for continuous experimentation. However, in spirit with the exploratory nature of the study, we also included other application types when our contacts mentioned their use of CD or continuous experimentation.

2.A.4 Case Study Procedure and Roles

The design of our study did not require direct access to specific company or project data (e.g., documentation, source code, test reports). The first round of interviews were conducted by the first, the second, and the fourth author, either on-site in the areas of Zurich and Vienna, or remotely via Skype. The deep-dive interviews (i.e., second round of interviews) were conducted by the first and the second author, either in Zurich, or remotely via Skype. The design of the quantitative survey involved all authors and the survey was hosted on the survey platform Typeform[7].

2.A.5 Data Collection

When starting the study, it was not decided how many iterations (i.e., steps) should be conducted. The initial design considered a single round of qualitative interviews and a quantitative online survey. To get more profound insights, we conducted a second round of interviews after the survey phase. Finally, data was collected using two rounds of interviews combined with a quantitative online

[7] `https://www.typeform.com/`

survey (i.e., data and methodological triangulation). Both data collection techniques include the direct involvement of software developers or release engineers, i.e., first degree contact according to Lethbridge et al. [2005]. The interviews with 31 developers of 27 companies are the primary source of information within the context of the study as much of the knowledge that is of particular interest (e.g., current issues with the release process) is not available anywhere else than in the minds of the interviewed participants. The quantitative survey was used to validate and substantiate the findings from the qualitative interviews.

Interviews

Interview design. Both rounds of interviews followed the same design. Based on the findings of the pre-study (i.e., key concerns and themes) an interview guide was generated that we used to conduct the first round of interviews. For the second round of interviews, we created a new questionnaire based on the results of the first round of interviews and the survey results to get more profound insights into the experimentation processes. For both interview phases, we fostered an exploratory character via a semi-structured interview process. All interviews included the mentioned themes and the discussion of each theme started off with an open question. Except for the first theme, topics were not covered in any particular order, but instead followed the natural flow of the interview.

Selection of participants. We recruited interviewees from industry partners and our own personal networks, and increased our data set using snowball sampling [Atkinson and Flint, 2001], i.e., by asking existing interviewees to put us in contact with further potential interview partners that they are aware of. Key factor for recruiting interview participants was that they have insights into the (technical) details of their company's or project's release process. Therefore, we refrained from interviewing participants in management roles. Another selection criterion was years of experience within the current company. We specified one year of experience as our lower limit for both rounds of interviews.

Survey

Survey design. To substantiate the findings of the first round of qualitative interviews, we designed an anonymous Web-based survey consisting of, in total, 39 questions. We structured the survey into the three themes release process in general, software deployment (covering the release and experiment evaluation, and roles and responsibilities key concerns), and issues in production (covering quality assurance, and issue handling key concerns). The survey mainly consisted of a combination of multiple-choice, single-choice, and Likert-scale questions. Although the survey had its focus on quantitative aspects, we also included some free-form questions to gain further thoughts and opinions in a more qualitative manner.

Survey participants. In surveys subjects are sampled from a population to which results are intended to be generalized [Runeson et al., 2012]. To address a "tech-savvy" population we distributed the survey within our personal networks (i.e., industry contacts), social media, via two DevOps related newsletters[8,9], and via a German-speaking IT news portal[10]. Survey participants reported an average of 8 years of relevant experience in the software domain (standard deviation 4 years). The resulting participant demographics for the survey is summarized on the right part of Figure 2.3.

Data Storage

All interviews were audio recorded with the interviewees' approvals. We sent a consent form to the interviewees multiple days prior the interviews containing details on data usage and storage. The audio files and the interview transcriptions generated during the process of data analysis will be stored for 5 years on a university server not accessible by the public. The recorded data will be properly deleted afterwards. The survey data will be exported from the survey platform and kept for five years on the same university server.

[8] http://www.devopsweekly.com/
[9] http://sreweekly.com/
[10] http://heise.de

2.A.6 Analysis

Coding. The first and the second author transcribed the recorded interviews. The first, the second, and the third author coded the transcriptions on a sentence level without a priori codes or categories. For the second round of interviews, we reused codes and added new ones when required. The free-form questions of the survey were coded following the same procedure.

Card sorting. The first three authors analyzed (i.e., investigator triangulation) the qualitative data using open card sorting [Spencer, 2009] (683 cards in total), and categorized the participants' statements, resulting in the set of findings presented in Sections 2.5 and 2.6. The cards were designed in such a way that each statement was on a single card supplemented with the participant's ID, the actual code, the company type (i.e., startup, SME, corporation), and the application type. The additional information was used to allow for better clustering and identifying differences amongst the various companies or projects involved. All clusters and thus findings are required to be supported by statements of multiple participants.

Chain of evidence. The pre-study was the basis for formulating the interview questions for the first qualitative phase. The card sorting findings of the first qualitative phase formed the basis for the survey questions and response options respectively (e.g., reasons against conducting experiments). We analyzed survey results using the statistical software *R*. The questionnaire of the second round of interviews was based on the analysis of survey results and the findings of the first qualitative phase. All the selected quotes in the paper represent coded statements.

2.A.7 Limitations

In Section 2.4 we present limitations and threats to validity associated with the single phases of our study in detail. An additional limiting factor throughout the interviews is that we only consider data points from a single perspective that are potentially biased having the participants providing idealized data about the CD and experimentation maturity of their companies. In the context of the study

it was not possible to analyze additional resources (i.e., data triangulation on a case level) such as process documentation, or deployment scripts. We tried to mitigate this factor by conducting a quantitative online survey to validate the findings of the first qualitative phase.

2.A.8 Reporting

Within this paper, we do not only report on the findings of our study, we also provide the reader additional information on the study design (i.e., this case study protocol). Moreover, we provide the interested reader a comprehensive online appendix[11] including all interview materials (i.e., questionnaires), survey questions, survey results in form of a report, and the survey's raw results. We do not expose the names of study participants and the companies they are working for. We used our findings to propose potential directions for future research to the research community.

2.A.9 Schedule

The first month of this research was used for planning the study, in the second month we conducted the pre-study. The first round of interviews required two months in total, the transcription of the interviews happened in parallel. Coding and card sorting took another month, similar to the preparation and execution of the survey. We spent a month on writing an initial version of this report. The second round of interviews was conducted in two months. The final data analysis (i.e., coding, card sorting) required us another month. A second version of this report was written afterwards (one month), which was revised three times since then (taking about a month each).

[11] `http://www.ifi.uzh.ch/en/seal/people/schermann/projects/expstudy.html`

3

Search-Based Scheduling of Experiments in Continuous Deployment

Gerald Schermann, Philipp Leitner
Published in the Proceedings of the 34th International Conference on Software Maintenance and Evolution (ICSME 2018)
Contribution: prototype implementation, data collection, data analysis, and paper writing

Abstract

Continuous experimentation involves practices for testing new functionality on a small fraction of the user base in production environments. Running multiple experiments in parallel requires handling user assignments (i.e., which users are part of which experiments) carefully as experiments might overlap and influence each other. Furthermore, experiments are prone to change, get canceled, or are adjusted and restarted, and new ones are added regularly. We formulate this as an optimization problem, fostering the parallel execution of experiments and making sure that enough data is collected for every experiment avoiding overlapping experiments. We propose a genetic algorithm that is capable of (re-)scheduling experiments and compare with other search-based approaches (random sampling, local search, and simulated annealing). Our evaluation shows that our genetic implementation outperforms the other approaches by up to

19% regarding the fitness of the solutions identified and up to a factor three in execution time in our evaluation scenarios.

3.1 Introduction

A high degree of automation (e.g., building, testing, and deploying software artifacts) enables companies, and especially Web-based companies, to release new functionality more frequently and faster. Companies such as Microsoft [Kevic et al., 2017], Facebook [Feitelson et al., 2013], Google [Tang et al., 2010], or Netflix [Gomez-Uribe and Hunt, 2016] are characterized by having hundreds of deployments a day throughout their software ecosystem. Sophisticated monitoring and telemetry solutions keep track of releases and the captured live production data has become the basis for data-driven decision making [Chen, 2015]. Instead of shipping new features or functionality to all users, continuous experimentation practices such as A/B testing [Kohavi et al., 2013] or canary releases [Humble and Farley, 2010] enable companies to test new features on a small fraction of the user base first. Fast insights from live data are paired with manageable risks, if things go wrong "just" a fraction of the users is affected.

However, setting up such an experimentation infrastructure is not a straightforward task, as demonstrated by experience reports of, e.g., Kevic et al. [2017], Xu et al. [2015], and Fabijan et al. [2017]. An essential requirement for successful experimentation is to collect enough data to draw valid statistical conclusions (cf. Kohavi et al. [2014]). Having multiple experiments running at the same time requires careful user assignments (i.e., which users are part of which experiments) as experiments might overlap and influence each other. To avoid these situations, some experiments might require to await the termination of previous experiments, or to run in parallel but on different fractions of the user base. The former is simple, yet not feasible as development work continues and delayed releases should be avoided. The latter requires scheduling a potentially scarce resource (i.e., users interacting with the application) in a domain prone to change. Experiments fail frequently, captured feedback gets integrated, and experiments are reiterated, i.e., re-executed after these adjustments. For example, at Microsoft

Bing [Kevic et al., 2017], 33.4% of experiments are ultimately deployed to all users, and experiments are iterated 1.8 times on average. Hence, scheduling experiments is not a self-contained task. Experiment restarts, reschedulings (e.g., different user group, or day), and cancellations (i.e., pre-scheduled resources become available for other experiments) need to be dealt with. Furthermore, the search space of how users can be assigned to experiments is massive.

In this paper, we define the problem of experiment scheduling as an optimization problem. Experiments should start as soon as the coding part of a feature is done, avoiding delays of multiple days or even weeks. Moreover, enough data has to be collected throughout the experiments, but at the same time we need to guarantee that the scheduled resources are distributed fairly to foster parallel experiment execution. For this, we propose a genetic algorithm that is capable of (re-)scheduling (running) experiments. We envision our approach to become an active part of a release or deployment pipeline [Humble and Farley, 2010], periodically (e.g., daily or even multiple times a day) updating the experiment schedule, accounting for experiment cancellations or restarts. The resulting schedule is then used to instrument the system for experiment execution, i.e., administering the states of dynamic feature toggles [Rahman et al., 2016] or traffic routing mechanisms [Schermann et al., 2016b]. We compare the capability of our genetic algorithm with the capability of other search-based approaches (random sampling, local search, and simulated annealing). To summarize, our main contributions are:

- The definition of experiment scheduling as an optimization problem.
- Implementations of a genetic algorithm (including custom definitions of crossover and mutation), random sampling, local search, and simulated annealing for this domain.
- An extensive evaluation showing that the genetic algorithm outperforms the other search-based approaches by up to 19% regarding the fitness score of the solutions identified and up to a factor of three in execution time.

Our tooling, source code, and evaluation data (i.e., example experiments and scripts) are available online [Schermann and Leitner, 2018] fostering experiment replication and extension.

3.2 Background

In the following, we provide background information on the different types of experimentation, typical considerations before launching an experiment, and what problem arise in the context of scheduling continuous experiments.

3.2.1 Types of Experimentation

Experimentation practices such as A/B testing [Kohavi et al., 2013], canary releases [Humble and Farley, 2010], or dark launches [Feitelson et al., 2013] give companies fast insights into how new features perform while keeping the risks manageable at the same time. Using the terminology of Schermann et al. [2018a,b], these experiments are categorized into two flavors: *regression-driven* and *business-driven* experiments.

Regression-driven experiments are used to mitigate technical problems (e.g., performance regressions) when testing new features, to conduct (system) health checks, and to test the scalability of the application's landscape on production workload. These experiments typically run from minutes to multiple days and mainly involve the practices of canary releases and dark launches.

Business-driven experiments guide different implementation decisions or variants of features from a business perspective (e.g., do customers appreciate this feature). These experiments typically involve A/B testing and run for multiple weeks or even months.

A common practice for both types of experimentation are gradual rollouts [Humble and Farley, 2010] in which the number of users assigned to an experiment is increased in a stepwise manner (e.g., increase from 2% of the Canadian users to 5%, then 10% and so on).

3.2.2 Ingredients of Experimentation

The core ingredients for all these types of experiments are the same. Once the responsible developer or analyst decides to launch an experiment, they need to have an understanding of (1) what to measure, i.e., the overall evaluation

criterion (OEC) [Fabijan et al., 2017; Kohavi et al., 2009], (2) how many data points to collect for being able to statistically reason about the OEC and thus the experiment's outcome (i.e., sample size), (3) which users to conduct the experiment on (e.g., different user groups, regions), and (4) when to run the experiment.

An OEC is highly domain dependent (e.g., units sold, number of users streaming videos) and represents a quantitative measure of the experiment's objective [Kohavi et al., 2009]. An essential decision is which users or user groups to consider for an experiment, thus who to assign to control and treatment groups. Users in treatment groups test new functionality, while control group users continue using the previous (stable) version and serve as reference points for (statistically) evaluating the experiment's outcome. Further factors involve that user groups might interact differently with a system (e.g., usage behavior of users paying for the service vs. users using it for free), that these groups are of different sizes, and that the time when an experiment runs may matter (e.g., time of the day, day of the week). Consequently, the duration of an experiment highly depends on these factors, plus all the other experiments that run at the same time on the same or overlapping user groups, i.e., there might be less traffic available for a single experiment in such a setting requiring longer experiment duration to collect a sufficient number of data points.

3.2.3 Uncertainty of Experimentation

Continuous experiments test new ideas. However, it is quite natural that not every idea ends up being successful. Experiments fail, failures are analyzed, and if there is a way to improve, experiments are repeated. The consequence is that scheduling experiments is not a one-time task. Resources budgeted for canceled experiments can be reused by other experiments, potentially reducing their overall execution time as the required sample size is reached faster (though still keeping in mind that there often exists a minimum duration to measure a certain effect).

Furthermore, dealing with traffic profiles in the context of experimentation is trying to estimate how the future might be while looking into the past. However,

we cannot foresee how a new feature changes the users interactions' with a system. For instance, a particularly well-received new feature may cause traffic to explode. Consequently, experiment schedules need to be periodically re-evaluated, and experiments may need to be extended, shortened, or postponed.

In our research, we take this "uncertainty" into account and provide an approach that is able to deal with rescheduling of experiments, frequently adjusting to changed (user) behavior, and results in experimentation schedules that support a "valid" (i.e., avoid overlapping experiments) execution of multiple experiments at the same time.

3.3 Related Work

We distinguish between related work on continuous experimentation in general, involving experience reports on how companies conduct experiments, and how search-based software engineering techniques have been used in similar domains and in the context of scheduling.

Continuous Experimentation. Research on continuous experimentation has gained traction recently. There have been experience reports specifically investigating the process, challenges, and characteristics of conducting experiments in an enterprise company setting. These reports involve for example work by Kevic et al. [2017] and Fabijan et al. [2017] looking at the process of Microsoft. Moreover, the work of Xu et al. [2015] and Tang et al. [2010] specifically covers how LinkedIn and Google approach handling multiple experiments in parallel. Different to our own and Google's approach, at LinkedIn experiments are fully overlapping by default as in most cases their tests are restricted to the UI level and run on different (sub-)parts of the system. Google uses a system of layers and domains to divide up the user space in order to avoid overlapping and conflicting experiments. However, the underlying assumption is that there is always enough user interaction available to collect enough data across all layers and subdomains, which may be true for Google, but less so for other companies with different patterns of user interaction and a smaller user base. Beside research focusing on single companies, there exist also multiple empirical studies, e.g., Lindgren

and Münch [2016] and Schermann et al. [2018b]. Still within the context of continuous experimentation, but more from a data science angle is the work of Kohavi et al. [2013, 2014, 2009] and Crook et al. [2009].

Search-based Software Engineering. Over the last years search techniques, and especially genetic algorithms, have become popular to tackle problems in multiple areas of software engineering, ranging from test data generation [Xanthakis et al., 1992], software patches and bug fixes [Weimer et al., 2009], refactoring [Romano et al., 2014], effort estimation [Sarro et al., 2016], defect prediction [Fu et al., 2016; Tantithamthavorn et al., 2018], to cloud deployment [Frey et al., 2013]. In the context of continuous experimentation, Tamburrelli and Margara [2014] formulate automated A/B testing as an optimization problem. They propose a framework that supports the generation of different software variants using aspect-oriented programming, the runtime evaluation of these variants, and the continuous evolution of the system by mapping A/B testing to a search-based SE problem. While they focus on the generation of the variants, our focus is on the actual execution of continuous experiments taking into account various constraints such as a limited number of users and avoiding overlapping experiments. Heuristics including genetic algorithms and simulated annealing have also been used in the context of scheduling resources. Wall [1996] provides a general overview of resource-constrained scheduling and describes scheduling as a dynamic problem, i.e., scheduling algorithms must be capable of reacting to changing requirements (e.g., when the availability of resources changes, or interruptions occur).

To the best of our knowledge, the problem of scheduling continuous experiments (i.e., a domain that is prone to change) has not yet been tackled.

3.4 Problem Representation

We formulate the problem of finding valid experimentation schedules as an optimization problem. We use weighted-sum as the parametric scalarizing approach [Deb, 2011] to convert multiple objectives into a single-objective optimization problem. Given a set of n experiments $E = \{E_1, E_2, \ldots, E_n\}$, the goal

Property	Type	Description	Example
ID	Integer	Unique experiment ID	1
Type	Enum	Business- or regression-driven experiment?	Regression
Min Duration	Integer	Minimum experiment duration in hours	240
Sample Size	Integer	Minimum required exp. sample size (RESS)	10,000,000
Priority	Integer	Experiment priority (≥ 0)	6
Preferred UGs	[String]	List of preferred user groups to test with	[3, 4]
Gradual	Boolean	Stepwise increase of assigned users?	True
Start Traffic	Integer	In case of gradual, # of users to start with	10,000

Table 3.1: Input data for experiments

is to identify a valid schedule S with maximal fitness, thus, maximizing the values of the problem's objectives. In this paper, we focus on scheduling experiments targeting a single service. First, we discuss how experiments and schedules are represented. Next, we discuss how the fitness of a schedule is defined and what defines a valid schedule.

3.4.1 Experiments

Scheduling experiments requires some basic information for every experiment E_i (summarized in Table 3.1). Besides a unique *identifier*, this involves the *type* of the experiment (i.e., business- or regression-driven experiment), the minimum experiment *duration* that is needed for measuring a certain effect, the *sample size*, thus how many data points to collect for an experiment, the *priority* of an experiment, and the *preferred user group* to test with.

When it comes to how experiments accumulate the required sample size in the course of their execution we distinguish between *gradual* and *constant* traffic consumption.

Constant consumption: Throughout its execution an experiment "consumes" a constant amount of traffic, i.e., the number of data points to collect at every hour x is defined as $c_x = \frac{samplesize}{duration}$.

Gradual consumption: Starting with a sample size t, the number of data points accumulated at every hour throughout an experiment with duration d increases in a stepwise manner. In our case, we rely on a simple linear function

with a positive slope, i.e., the consumption c_x at hour x corresponds to $c_x = kx + t$. The factor k for the increase is obtained from the following integral:

$$\int_1^d kx + t\,dx = samplesize$$

Thus, the total consumption throughout the experiment (i.e., the area of the linear function for the interval 1 to d) has to sum up to the minimum sample size.

3.4.2 Schedules

A schedule S for n experiments $E = \{E_1, \ldots, E_n\}$ consists of a set $S = \{S_1, S_2, \ldots S_n\}$, in which every schedule S_i corresponds to an experiment E_i. Every schedule S_i is a tuple $\langle \tau, A \rangle$, consisting of a start slot τ (i.e., the hour to launch the experiment) and a tuple of assignments $A = \langle A_1, A_2, \ldots, A_d \rangle$, where d corresponds to the duration of the experiment in hours. Every individual assignment A_i specifies how many users (in percent) of which user groups are part of the experiment at hour i. This is defined as a matrix $\{group : consumption\}$, e.g., $A_4 = \{group1 : 0.05, group2 : 0.0, group3 : 0.01\}$. In this example, 5% of the users of user group 1 and 1% of the users of user group 3 are part of the experiment at hour 4 of its execution.

3.4.3 Fitness

The fitness function applied on a schedule S represents a trade-off between three conflicting objectives: experiment duration, start, and user group. All three are assigned separate scores.

Duration score: An experiment E_i should not take longer than required, i.e., the length of its schedule S_i should – in the best case – equal the experiment's minimum duration. For example, results for an experiment on the system's scaling capabilities should be available after 3 days instead of 2 weeks. The duration score ds_i of experiment E_i corresponds to $ds_i = \frac{minDuration}{d}$, where d denotes the duration of the schedule S_i. Thus, the maximum score equals 1.0

if and only if the experiment does not take longer than its specified minimum duration.

Start score: Experiments should start as soon as possible. The start score ss_i of experiment E_i with schedule S_i corresponds to $ss_i = \frac{1}{\tau}$. Thus, the maximum score equals 1.0 if and only if the experiment starts at hour $\tau = 1$.

User group score: An experiment E_i should (mainly) involve its *preferred user groups* during its execution. The user group score us_i of experiment E_i corresponds to $us_i = \frac{\sum_1^d coverage(A_i)}{d}$, in which $coverage(A_i)$ is a function returning 1.0 if and only if at least one of the experiment's preferred user groups captures the majority of the sample data at hour i, otherwise 0. The sum is then divided by the experiment's duration d, resulting in the average user group coverage. Consequently, if the coverage criterion is fulfilled for every hour of the experiment's execution, the maximum score equals 1.0.

Combined fitness score: The combined fitness score f of schedule S consisting of n experiments $E = \{E_1, \ldots, E_n\}$ is obtained by summing up the individual scores for every experiment E_i, taking into account the different experiment priorities $\langle p_1, \ldots, p_n \rangle$ and weighting of the scores $\langle w_{ds}, w_{ss}, w_{us} \rangle$. Therefore, we transform our three objectives into a single scalar objective using the weighted-sum strategy, thus leading to a single-objective optimization problem. The weights sum up to 1, thus, the fitness score of a schedule S is in the range 0 to 1. Prioritization allows favoring some experiments over others.

$$f = w_{ds} * \frac{\sum_1^n ds_i * p_i}{\sum_1^n p_i} + w_{ss} * \frac{\sum_1^n ss_i * p_i}{\sum_1^n p_i} + w_{us} * \sum_1^n us_i * p_i$$

3.4.4 Constraints

For a schedule S to be considered valid, four constraints have to be fulfilled. We distinguish between experiment constraints and overarching constraints. The former checks on experiment level (i.e., checking S_i of E_i), the latter requires checking the entire schedule S. Experiment constraints involve checking for valid business-driven experiments, checking that experiments collect sufficient data,

and that they are not interrupted, the overarching constraint ensures that we schedule not more resources than available.

Valid business experiments: A schedule S_i of a business-driven experiment E_i (i.e., experiment type = `business`) is valid if and only if it involves the same user groups during all hours of its execution, i.e., the user groups with *consumption* ≥ 0 in every $A_j \in A$ of the schedule S_i's assignment A are the same. The reason for this constraint is that business experiments measure a certain effect for particular user groups, often for a long period, therefore switching user groups during the execution would skew results. However, for a regression-driven experiment testing, for example, the scaling capability of a new service, it generally does not matter whether user groups change within the experiment.

Sufficient data points: This constraint validates that *consumedTraffic*(A_x) $\geq c_x$ for every hour $x = \langle 1, \ldots, d \rangle$ of the experiment E_i with schedule S_i, duration d, and assignments $A = \langle A_1, \ldots, A_d \rangle$. The function *consumedTraffic*(A_x) returns the total number of users that are assigned to the experiment at hour x taking into account the traffic that is expected according to the underlying traffic profile (e.g., see an example profile for a user group in Figure 3.3). Thus, it is checked whether the minimum required sample size c_x (either constant or gradual) is met for every hour.

Non-interrupted experiments: This constraint ensures that experiments continuously collect data throughout their execution. Thus, there does not exist an assignment A_i consuming zero traffic and there has to be an assignment $A_i \in A$ for every $i = \langle 1, \ldots, d \rangle$.

Sufficient traffic available: This overarching constraint ensures that at a time slot x there is no user group across all schedules $S = \{S_1, \ldots, S_n\}$ such that the total traffic consumption within a user group is more than 100%. For every schedule S_i we consider the user groups of assignment A_j with $\tau + j - 1 = x$.

3.5 Approach

In this section we look at approaches to generate valid experiment schedules as solutions for the presented optimization problem. First, we start with a genetic

algorithm, followed by random sampling, local search, and we conclude with simulated annealing as a slight modification of local search.

3.5.1 Genetic Algorithm

Genetic algorithms (GA) [Fonseca and Fleming, 1993] group candidates, typically called individuals, in populations. The basic idea of genetic algorithms is to mimic an evolutionary process in which the best-suited candidates in each generation are selected and used as basis for the next generation of solutions. Starting with an initial population of multiple individuals (i.e., different valid solutions of the optimization problem), these selected candidates evolve through multiple generations in which mutation and crossover operations are applied and after several generations of reproduction those individuals that inherited superior properties become dominant. The reproduction within each generation consists of the following basic steps for genetic algorithms as presented, for example, by Harman [2011]. In our case an additional *repair* step is added (see Section 3.5.1).

1. Parent selection
2. Crossover
3. Offspring mutation
4. Repair
5. Fitness and validity evaluation
6. Next generation selection

The initial population is created using random sampling (see Section 3.5.2 for details). Individuals are represented as chromosomes (see Section 3.5.1) and a fitness function serves as basis for their assessment (see Section 3.4.3). Individuals for the next generation are primarily created using crossover and mutation operations, but a small set of the best individuals (*ELITISM_SIZE* parameter) of the previous generation is also passed on to the next generation unchanged (step 6). The GA stops after a specified number of generations, or if one individual solution reaches the desired level of fitness.

All of the presented approaches (i.e., genetic algorithm, local search, simulated annealing) use random sampling as starting point and the chromosome structure described in the following to represent solutions for the optimization problem.

Chromosome Representation

A chromosome, i.e., a solution of the optimization problem, is represented as shown in Figure 3.1. We rely on value encoding and a chromosome corresponds to a schedule S defined in Section 3.5, i.e., a chromosome consists of multiple schedule genes $S_i \in S$ corresponding to their experiments $E_i \in E$ (top layer in Figure 3.1). Every schedule gene S_i further contains assignment genes $A_j \in A$ and a gene encoding the start hour of the schedule S_i (middle layer in Figure 3.1).

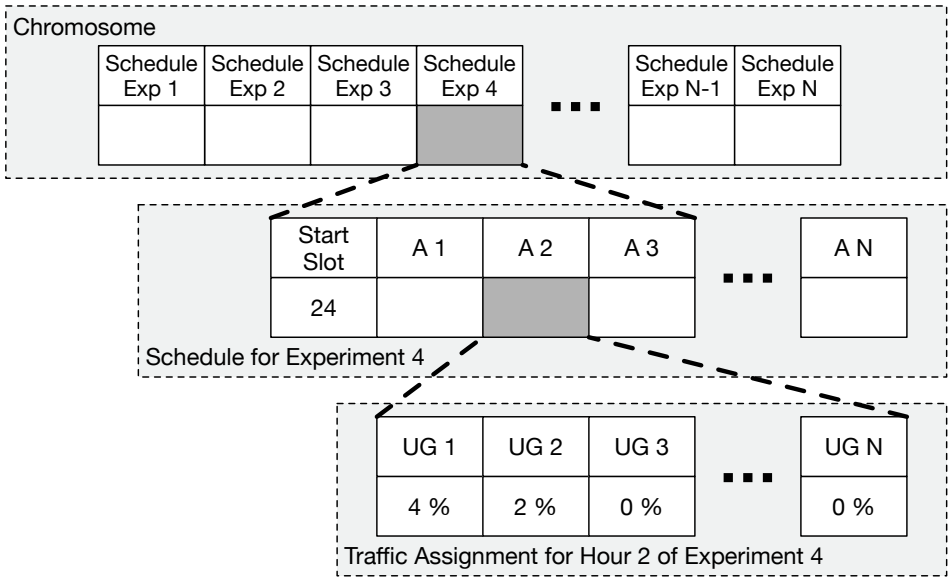

Figure 3.1: Chromosome representation using value encoding

Every assignment gene further contains a gene for every single user group (UG). This is used to encode how many users of a certain user group are assigned to the experiment E_i at the respective hour. In Figure 3.1, bottom layer, at hour

25 (start time slot (τ) $24 + 2 - 1$), 4% of the users of user group 1 are assigned to experiment 4.

Parent Selection

The selection of individuals (i.e., parents) for reproduction within each generation plays an important role. Selecting only the highest-score individuals could result in reduced genetic diversity and lead to premature convergence, thus there is the chance that the entire population gets "stuck" at a lower quality solution. Hence, we use fitness proportionate selection [Goldberg and Deb, 1991], as it is simple to implement and has been proven to produce acceptable solutions [Bäck et al., 2000]. The fitness of every individual is obtained, those individuals with higher fitness have a higher probability to get selected for reproduction (i.e., crossover and mutation steps).

Crossover

Crossover is the process of creating an offspring of two selected parent individuals by applying a crossover operation with a certain probability P_c. In our implementation, when performing a crossover of two individuals A and B, we compare the fitness on experiment level. Thus, for each experiment $E_i \in E$, we compare the fitness of A's schedule S_{iA} with the fitness of B's schedule S_{iB}. The single schedule with the higher fitness is added to the offspring as visualized in Figure 3.2. Consequently, in our approach a single offspring is created during crossover and moved to the subsequent mutation step. In case that no crossover happens (regulated by P_c), both parents are passed on unchanged to the mutation step.

Mutation

Once the offspring is created by applying the crossover operation, the offspring is mutated by performing NUM_OPS mutation operations on randomly selected experiments E_i and their respective schedule S_i with a mutation probability P_m. Mutation is important to ensure genetic diversity within the evolving population

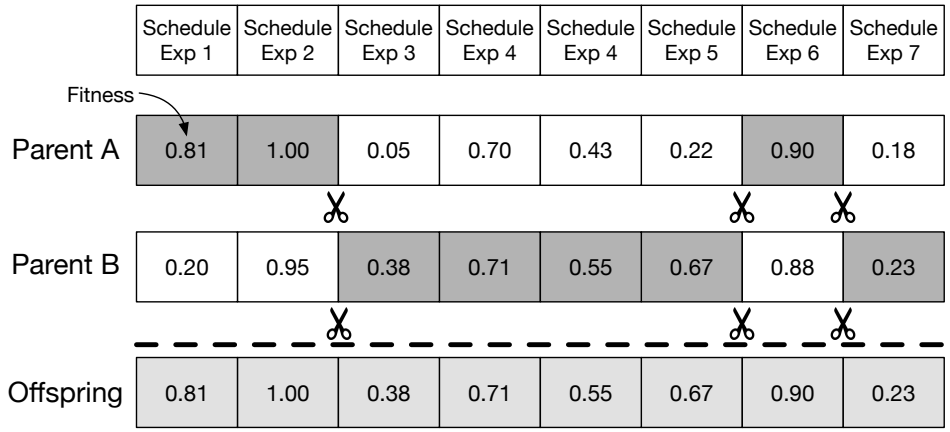

Figure 3.2: Crossover example

and helps to avoid convergence to a local optimum. In our implementation, six mutation operations exist that are described in the following.

Move schedule: Pre- or postpones the execution of a certain experiment E_i by $MOVE$ hours by mutating the start slot τ in the respective schedule S_i. In case of a reevaluation of the entire experiment schedule, i.e., experiments are already running, some get canceled, and new ones are added to the schedule, those experiments that are already running are omitted for move operations as this would violate the non-interruption constraint.

Shorten/Extend schedule: Shortens (extends) the selected schedule S_i of experiment E_i by $SHORTEN$ (EXT) hours. In case of shortening, $SHORTEN$ assignment genes are removed from S_i's assignment. In case of extension, EXT assignment genes are added to S_i's assignment by duplicating the last assignment gene.

Flip user group: Changes which user groups are assigned to the selected experiment E_i, either for the entire schedule S_i (i.e., all $A_i \in A$), or for a certain range (u, v) within the schedule (i.e., $A_k \in A, 1 \leq u \leq k \leq v \leq d$). Takes randomly a used user group ug_x of the experiment's schedule S_i (or from within the range) and replaces it with another randomly retrieved user group ug_y. If the new user group ug_y was already used within S_i, the traffic of ug_x is added to ug_y's traffic. In case of schedule reevaluation, already running business-driven

experiments are excluded from the flip operation. In addition, the flip range operation is not applied on business experiments as this would lead to a constraint violation.

Add/Remove user group: Similar to *flip user group*, instead of replacing a user group for the entire schedule S_i (or for a certain range within the schedule), an unused user group is added, or a used user group is removed as long as there is at least one user group left. The same conditions apply for schedule reevaluation involving running business experiments and range operations on business experiments.

Repair

Mutating a schedule S_i of an experiment E_i has a direct effect on the number of data points collected during its execution. In order to ensure that enough data points are collected to fulfill the validity constraints (e.g., after a user group is removed from the experiment), a *repair action* is executed after the mutation step. The repair action adjusts for every mutated schedule S_i every single assignment $A_j \in A$ in such a way that $consumedTraffic(A_j) \geq c_j$ is fulfilled. The repair action distributes the required data points for every c_j across the user groups used by A_j. This is achieved by considering the estimated traffic within the user groups at a specific time slot x from the underlying traffic profile and the required sample size for this specific hour c_x which itself depends on the experiment's sample size and its schedule's duration. To deal with the uncertainty regarding the estimated traffic profile, we introduce a buffer ($BUFFER$ parameter) such that slightly more (e.g., 0.5%) traffic is consumed than required.

3.5.2 Random Sampling

Random sampling (RS) [Schoen, 1991] tries to find valid solutions by creating individuals purely by chance. Our RS approach makes use of the fitness function for assessing the individuals and the constraints for checking their validity as presented in Section 3.4. A set of POP_SIZE individuals for n experiments $E = \{E_1, \ldots, E_n\}$ to be scheduled is created as follows. For an individual solution,

randomly take an experiment E_i to be scheduled. Create a schedule S_i with a random start time, a random selection of one or two user groups out of the pool of existing user groups, and a random duration d that is larger or equal than the experiment's minimum duration. Then create d assignment genes such that the minimum required sample size of E_i is reached during the course of the experiment on the selected user groups on the estimated traffic profile. If the created schedule S_i is valid, then it is added to the overall schedule S and the next experiment E_j is picked for scheduling. The random schedule creation is repeated until a valid schedule for every picked experiment is found. After *POP_SIZE* valid individuals are created, the one individual with the highest fitness score is chosen as the result of RS. Strictly speaking, as we pick one experiment after the other and only proceed when a valid schedule is found, our approach is not "pure" random sampling but rather categorized as systematic random sampling [Schoen, 1991].

3.5.3 Local Search

The local search (LS) algorithm starts with the best individual generated by random sampling and iteratively tries to optimize it by applying the same mutation operations as with the genetic algorithm, followed by the same repair step after each iteration. Again *NUM_OPS* mutation operations are performed on randomly selected experiments E_i and their respective schedule S_i. If the newly generated neighbor resulting from the mutation is an invalid solution, the mutation is reset and the process is repeated until a valid solution is found. After this step, the resulting valid neighbor is compared to the current solution, if the neighbor's fitness score is higher, then the neighbor becomes the current solution. This process is then repeated *NUM_ITERATIONS* times and the final "current" solution returned.

3.5.4 Simulated Annealing

The main issue of local search algorithms is that — by design — they get stuck in a local optimum from which no further improvements are possible. Simulated

annealing (SA) [Schoen, 1991] as a variant of local search algorithms tries to overcome this issue by applying a technique simulating the physical process of annealing in metallurgy. Transferred to our optimization problem and in contrast to our local search implementation, neighbor solutions with worse fitness than the current solution have a certain probability to get accepted, thus the likelihood to run into a local optimum is reduced. The likelihood to accept worse solutions is tied to the current temperature. Initially the temperature is high, thus the algorithm is more likely to accept neighbor solutions with a lower fitness score than the current solution. After every iteration the temperature slowly decreases by a cooling factor, thus the acceptance of worse solutions is less likely. The process of finding valid neighbors and optimizing them using the mutation operators is exactly the same as in our local search implementation, just with the slight addition that the acceptance criterion is added.

3.6 Evaluation

For the evaluation of the capabilities of the previously discussed approaches, we implemented them in Java and we assessed them in three aspects: (1) maximum fitness scored for a specific set of experiments, (2) comparison when running an increasing amount of experiments at the same time, and finally, (3) dealing with the reevaluation of an existing schedule. Before we dive into the evaluation, we briefly describe the setup we used as basis for the evaluation.

3.6.1 Setup

The setup involves the description of the used traffic profile, the experiments created (in different sizes), how the various algorithms were calibrated, and finally, on which hardware we executed the evaluation runs.

Traffic Profile

For our evaluation we mimicked a real traffic profile. We used GitLab's public monitoring tooling[1] and extracted the hourly interaction of users based on the number of returned HTTP status codes for the months January and February 2018. This total traffic per hour served as our baseline and we reserved 10% traffic to serve as the control group being not involved in any experiment. For our evaluation scenario, we divided the remaining traffic into five user groups: group 1 (40% traffic, simulating logged off users), group 2 (10%, paying single license users), user group 3 (20%, free single users), user group 4 (15%, paying company license users), and group 5 (15%, free company users). To simulate longer running experiments we replicated the two month period to get a twelve month profile.

Experiments

We created a baseline of 10 experiments. This involved six regression-driven experiments (two with gradual, four with constant consumption) and four business-driven experiments (constant consumption). Their minimum duration ranged from a single day up to 18 days. As basis for our experiments we used the durations reported by Kevic et al. [2017] for Microsoft Bing. To evaluate the algorithms under different scenarios, we created three variations of our baseline: with low, medium, and high required experiment sample sizes (*RESS*), i.e., how many data points does an experiment need to collect to reason about a certain effect. The baseline with low *RESS* requires 15 million data points in total (i.e., the sum of the *RESS* of the 10 experiments), with medium *RESS* 30 million, and with high *RESS* 55 million.

To evaluate the algorithms on different numbers of experiments running in parallel, we used the baseline of 10 experiments and duplicated them with a step size of 5 experiments to create sets with up to 70 experiments. This resulted in sets of $10, 15, 20, 25, \ldots, 70$ experiments, each set in 3 variations with low,

[1] `https://monitor.gitlab.net/dashboard/db/fleet-overview`

medium, and high *RESS*. For example, 70 experiments with high *RESS* require to collect $55 * 7 = 385$ million data points in total.

Figure 3.3 demonstrates the effect of the different *RESS* variants when scheduling 30 experiments. The high number of required data points leads to a longer schedule in case of the high *RESS* variant. There is simply not enough traffic available within user group 3 to host all experiments in parallel at the same time. Further, Figure 3.3 depicts the effect of the gradual experiments and how the numbers of users assigned to these experiments increase during their execution.

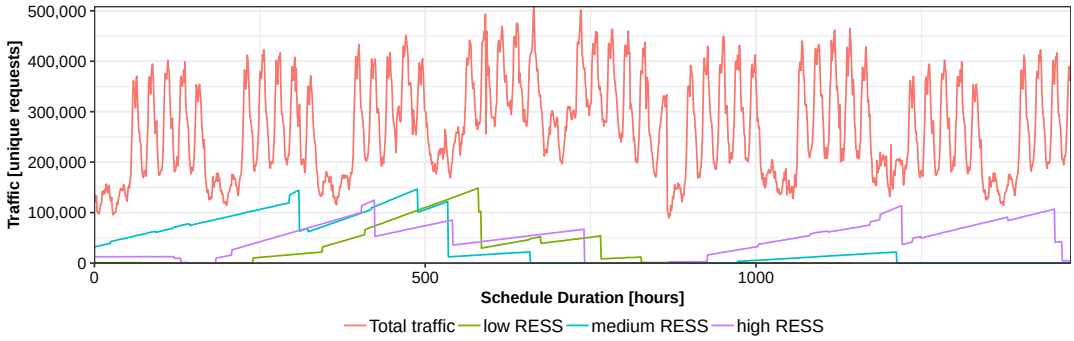

Figure 3.3: Traffic profile for user group 3 and traffic consumption of three example schedules (30 experiments each) with low, medium, and high *RESS*.

Calibration

To calibrate the algorithms we followed an iterative exploratory parameter optimization procedure with 25 experiments with low *RESS*. We increased the population size and the number of generations for the GA starting from 10 in steps of 10 until we reached 100. The number of iterations for LS and SA was evaluated for 1,000 to 10,000 iterations (step 1,000). Crossover probability was increased from 70% to 100% (step 5%), and mutation probability from 10% to 100% (step 10%). The number of the executed mutation operations *NUM_OPS* is defined as being dependent on the number of experiments to schedule (e.g., 10

experiments and *NUM_OPS* of 20% leads to 2 mutation operations). We tested *NUM_OPS* from 5% to 30% (step 5%).

Based on this procedure, we decided for a population size of 40, 90 generations, a crossover probability of 90%, a mutation probability of 50%, *NUM_OPS* of 15%, *ELITISM_SIZE* of 5, and 3000 iterations (LS and SA). For the sample with 25 experiments, SA achieved the best results with a starting temperature of 0.007 and temperature decrease of 1% per iteration. We further set *MOVE* to 48 hours, *SHORTEN* and *EXT* to 6 hours. For obtaining a scalar fitness value we used the weightings $\langle w_{ds} = 0.4, w_{ss} = 0.4, w_{us} = 0.2 \rangle$.

Hardware

We conducted the evaluation on the Google Compute Engine public cloud service. We used custom Intel Skylake instances with 4 vCPUs and 4.75 GB memory running Debian 9 and OpenJDK 8.

3.6.2 Maximum Fitness

Given a set of 15 experiments with medium *RESS*, the goal of this aspect of evaluation is to identify the maximum fitness score we can obtain for any of the discussed algorithms. Further, we want to identify how stable the results are, i.e., we repeat the execution of each algorithm 20 times. In contrast to the other aspects of the evaluation (i.e., stepwise increase and reevaluation) that exactly use the findings of the calibration, we use 150 generations for the genetic algorithm (GA) and 5000 iterations for local search (LS) and simulated annealing (SA). The reason is to give the algorithms more time to optimize their results, while the calibration results (90 generations, 3000 iterations) were a trade-off between fitness score and execution time, especially taking effect for the stepwise evaluation.

To have a fair comparison of the algorithms, for every of the 20 repetitions, we create an initial population using RS that is then also used for the respective runs of the GA, LS, and SA. We measure the execution time for every run, including the time it takes to generate the initial population.

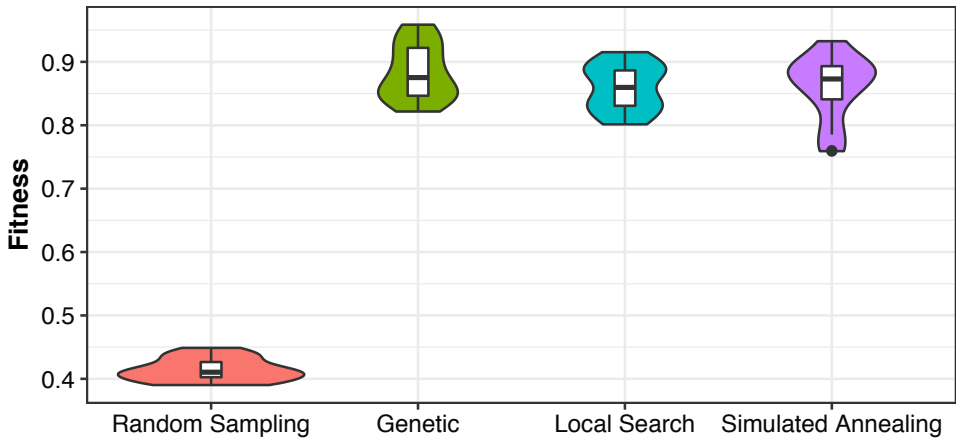

Figure 3.4: Comparison of fitness scores for 15 experiments to schedule (GA: 150 generations, LS & SA: 5000 iterations), 20 repetitions in total

Statistic	RS	GA	LS	SA
Mean exec. time (min)	0.96	19.88	26.64	26.04
Mean fitness	0.41	0.88	0.86	0.86
Max fitness	0.45	0.96	0.92	0.93
SD fitness	0.02	0.04	0.04	0.05
Mean *ds*	0.67	0.97	0.84	0.85
Mean *ss*	0.02	0.74	0.82	0.81
Mean *us*	0.70	1.00	0.99	0.99

Table 3.2: Statistics for Scheduling 15 Experiments with Medium *RESS*

Figure 3.4 visualizes the resulting fitness scores in form of violin plots. Table 3.2 provides additional statistics. The GA, LS, and SA implementations optimize the fitness score of RS by a factor ≥ 2. GA achieves slightly better fitness scores than LS and SA in less execution time (20 vs. 26 minutes). The achieved fitness scores are relatively stable with a SD of about 4%. Breaking down the combined fitness score into the individual scores (duration score *ds*, start score *ss*, and user group score *us*) reveals that the GA reaches almost the absolute minimum duration (*ds* 97%), while LS and SA are better when it comes to letting the experiments start as early as possible (*ss* ∼80%). When analyzing

how the scores for the best run (i.e., max fitness) evolve, we notice that in case of GA, no further optimizations are performed after 140 generations, in case of LS, the scores are stable after 4800 iterations, and only in case of SA, the scores keep changing even at iteration 5000.

3.6.3 Dealing with Multiple Experiments

The goal of this aspect of our evaluation is to identify how the algorithms deal with an increasing amount of experiments to schedule. We conduct evaluation runs in a stepwise manner. Starting with 10 experiments, we increase the number of experiments to schedule by 5 experiments per step, until we reach 70 experiments. Similar to the previous aspect, we are primarily interested in the fitness scores achieved, the overall execution time, and how the single objectives evolve. For every step, we conduct 5 runs with low, medium, and high *RESS* each, i.e., 15 runs per algorithm per step. Again, the GA, local search (LS), and SA implementations use the initial population generated by RS for the respective runs. We use the parameters determined by the calibration runs.

Figure 3.5 visualizes the fitness scores achieved and the error bars (± one standard deviation) combining the results of the runs with low, medium, and high *RESS*. In addition, Table 3.3 outlines execution behavior, i.e., how long it took to generate the schedules. For space reasons, we omit the results on medium *RESS*. As a single run for LS and SA took up to 10 hours for 45 experiments (with high standard deviations), we decided to cut the evaluation at this point.

The GA outperforms the other approaches for 20 and more experiments to schedule. This does not only apply for the achieved fitness scores (e.g., 40 experiments with high *RESS*: GA reaches 62%, SA 42%, and LS 43%), but also when it comes to execution behavior. While it takes the GA on average 110 minutes to schedule 40 experiments with high *RESS*, LS and SA take almost three times as long on average (280 and 274 minutes). The GA implementation was able to finish scheduling 70 experiments (with low *RESS*) within 8 hours. Similar to the previous evaluation, the achieved fitness scores are quite stable. The error bars in Figure 3.5 are mainly driven by the slightly different results (± 5 – 6%) of the runs with low, medium, and high *RESS*. Runs with low *RESS*

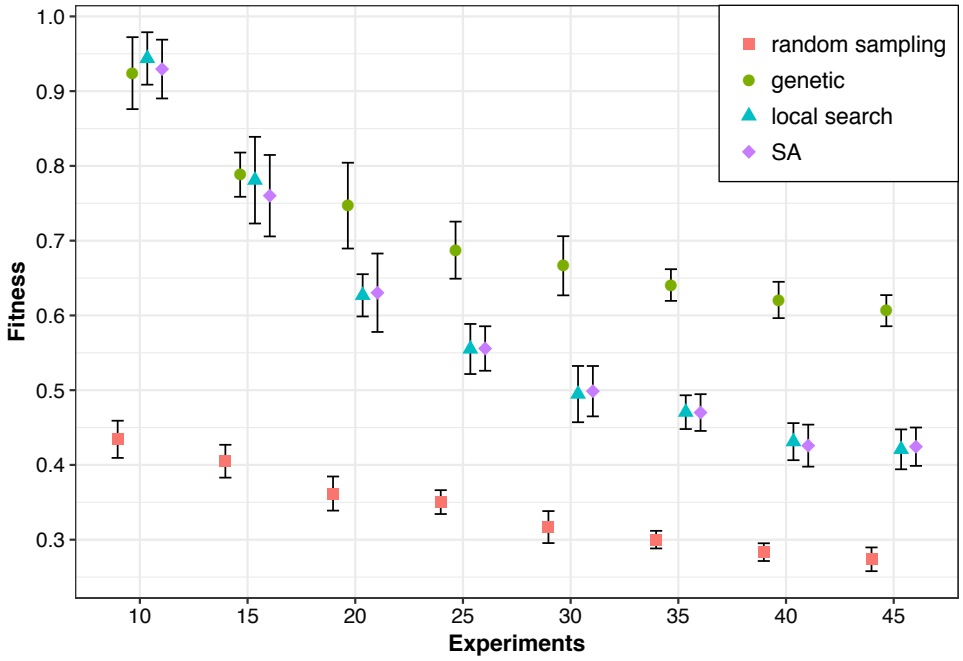

Figure 3.5: Fitness scores obtained for different algorithms when number of experiments to schedule is increased. Error bars represent ± one standard deviation.

achieve better results. Inspecting only runs within a certain stepsize and within the same *RESS*, the standard deviation of the achieved fitness scores is rarely larger than 3%.

Breaking down the fitness score into individual scores, we notice that the user group score (i.e., scheduling on preferred user groups) is in almost all cases $\geq$ 98% (except random sampling in which no optimization happens). The GA is strong when it comes to keeping the experiment's execution duration short, the duration score *ds* is on a high level throughout the various step sizes and decreases only from 98% when scheduling 10 experiments to 83% when scheduling 45 experiments with high *RESS*. In contrast, SA and LS reach a *ds* of 82% when scheduling 15 experiments and the score drops below 50% when scheduling 40 and more experiments (for all *RESS* variants). Similar to our previous observation, LS and SA begin with higher start scores (i.e., running more experiments right at

		Number of Experiments							
	Stat.	10	15	20	25	30	35	40	45
RS low	Mean	0.1	0.7	1.6	3.7	6.6	16.4	18.1	42.5
low	SD	0.0	0.0	0.1	0.4	0.6	3.4	0.9	14.4
high	Mean	0.2	1.1	2.3	5.0	9.4	14.5	25.7	43.9
high	SD	0.0	0.1	0.1	0.3	0.4	1.0	3.4	14.3
GA low	Mean	2.9	9.5	14.2	26.4	36.9	69.7	74.4	129.1
low	SD	0.2	1.2	0.5	1.4	2.3	10.6	5.0	23.9
high	Mean	5.5	14.5	24.3	45.4	60.6	86.1	110.5	178.5
high	SD	0.7	0.9	1.6	1.6	4.8	6.6	6.8	21.0
LS low	Mean	3.9	14.9	32.0	54.9	93.9	168.4	204.3	517.2
low	SD	0.7	1.7	3.4	5.0	6.7	18.0	13.1	321.6
high	Mean	6.6	20.9	47.6	103.2	153.0	194.3	280.2	416.5
high	SD	1.3	3.4	8.2	10.6	28.2	28.2	38.3	46.7
SA low	Mean	3.9	13.8	32.4	57.6	92.6	169.9	204.7	586.2
low	SD	0.6	1.4	1.2	2.8	4.3	7.9	22.4	355.6
high	Mean	7.7	21.1	49.9	104.2	159.2	200.0	273.7	453.8
high	SD	2.5	3.0	9.7	16.2	34.0	31.3	27.1	126.3

Table 3.3: Comparison of Exec. Times in Minutes for Increasing Number of Experiments to Schedule with Low and High *RESS*

the schedule's launch), but with an increasing number of experiments to schedule the scores drop below 25% with 25 experiments or more. The start scores of the GA are worse in the beginning (e.g., 10 and 15 experiments), but the decline throughout the various stepsizes is smaller.

3.6.4 Reevaluating an Existing Schedule

One essential requirement for our approach is that the implementations are able to deal with the reevaluation of a schedule, i.e., taking into account experiments that (1) finished within the already executed period, (2) got canceled, and (3) are added to be scheduled as well. Further, reevaluation means that the required sample sizes of running experiments (i.e., *RESS*) are adjusted according to the actual data points that were captured until the moment of the reevaluation. Thus, depending on the real traffic situation, an experiment's duration might need to be adapted.

To test this behavior of our implementations, we select the best resulting schedule of the GA from the previous evaluation step with 30 experiments and medium *RESS*. The schedule's fitness value is 74% (duration score 88%, user group score 100%, start score 47%). The reevaluation is conducted after 72 hours. Out of the initial 30 experiments, 3 are canceled, 3 finished within the 72 hours, and 5 new experiments with medium *RESS* are added.

We conduct 10 evaluation runs in total. Again, the resulting population of every run of RS is used by the respective GA, LS, and SA runs. Random sampling in the context of a reevaluation is special as one individual within the population is based on the existing schedule, taking into account already existing optimizations. For the newly added experiments within this individual the usual sampling process applies.

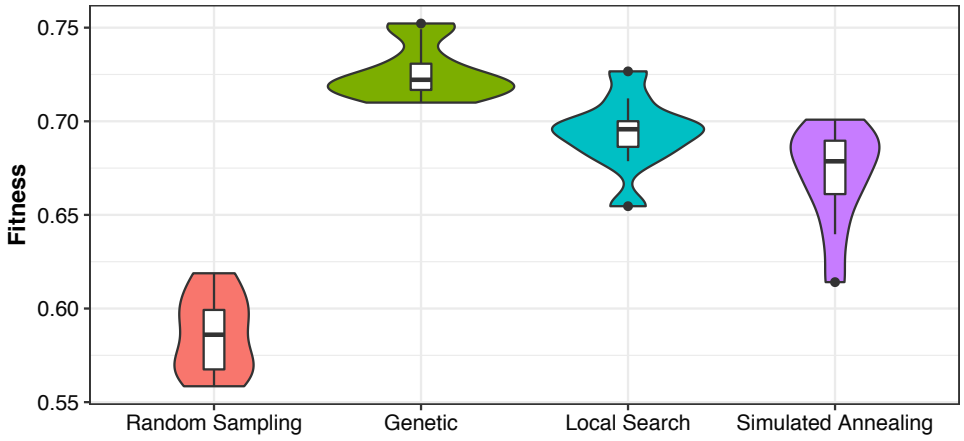

Figure 3.6: Fitness scores after reevaluation (10 runs): (re-)scheduling 29 experiments in total (3 discarded, 3 finished, and 5 new experiments)

Figure 3.6 again shows the achieved fitness scores using violin plots. The fitness scores of the resulting schedules are slightly below (1% in case of GA) the values of the original schedule, but still in a similar range as the evaluation runs of the previous aspect. The gap between the individual approaches (i.e., GA 73% fitness on average, LS 69%, and SA 67%) is much smaller than in the previous evaluation runs with a similar number of experiments to schedule (e.g.,

30 experiments, medium *RESS*: GA 68% on average, LS 50%, SA 51%). The reason is that both LS and SA benefit from an already optimized schedule with an especially high duration score *ds*. The execution time is on a similar level than for scheduling 30 experiments: 29 minutes on average for the GA, and 52 minutes on average for both LS and SA.

3.7 Discussion

We now briefly reflect on and discuss the implications of our results.

Scheduling as Part of a Release Pipeline. As observed during our evaluation, for a smaller number of experiments (i.e., 10 and 15) the achieved fitness scores of the GA, LS, and SA implementations are on a similar level. However, when it comes to a larger number of experiments, the GA implementation not only outperforms the other approaches in the fitness scores, but also drastically in execution time. This is especially of interest when we envision scheduling and (re-)scheduling of already running experiments (which achieves stable results as demonstrated in the final aspect of our evaluation) to become an active part in a release pipeline, e.g., the scheduling is triggered as soon as source code changes pass the quality assurance phases. Clearly, the maximum acceptable execution time for scheduling to become part of a release pipeline depends on the release frequency of a company, but for example an execution time of 40 minutes to schedule a set of 30 experiments on cheap public cloud instances is a promising result. Further, due to the nature of the genetic algorithm (i.e., offspring for the next generation is created independently) a higher level of parallelization is possible compared to the LS and SA implementations. Thus, we expect that stronger computing machinery could even decrease the time needed to find suitable solutions.

Importance of Calibration. It is not a straightforward task to tune multiple parameters to achieve acceptable results in various execution scenarios. This can be especially observed for our results of SA. Even though there are "only" two parameters to tune, in most of our evaluation runs SA performed slightly worse than its counterpart local search. The reason is that the starting

temperature and the cooling factor were calibrated for a set of 25 experiments. Consequently, we would need to fine-tune these parameters for different numbers of experiments to achieve better results. Another factor, not only influencing the results of SA, is the weighting of the three fitness scores (i.e., objectives). We have observed that the user group score achieves very high values (rarely below 98%) across our evaluation runs. This could be an indicator that the weighting could be decreased to better optimize for the other two objectives. In cases for which scheduling the preferred user group is of absolute importance this could be ensured by choosing a higher experiment priority.

Crossover and the Destruction of Valid Schedules. One of the important steps that help genetic algorithms avoiding the traps of a local minimum or maximum is its crossover operation. In our implementation, crossover returns a single offspring and this offspring is created in a "greedy" way. An potential effect of this setup on the GA's results is that the duration scores are consistently higher compared to LS and SA implementations. The downside of this approach is that during the process of reproduction the validity of the schedule and thus the overarching constraints are not taken into account. Consequently, many created children are invalid and thus thrown away. We experimented with multiple different strategies, conservative ones such as coin flips on whether to include a schedule for an experiment from the first or second parent, and "smarter" ones such as trying to preserve how user groups are distributed. However, all of these alternative strategies were outperformed by the "greedy" variant. However, we still believe that there is space to improve how offspring is created during the crossover process by better taking validity constraints into account.

3.8 Threats to Validity

We now the discuss issues that form a threat to the validity of our results.

Construct Validity. The main threat regarding construct validity is that our definition of scheduling continuous experiments as an optimization problem is not an adequate representation of the domain. This is especially the case for the definition of constraints (i.e., the validity of a schedule) and how we determine the

fitness of a schedule (e.g., fitness function and the used weighted-sum approach). We mitigated this threat by strongly relying on reported empirical work (e.g., Kevic et al. [2017], Schermann et al. [2018b], and Lindgren and Münch [2016]). Another threat regarding the representation is that we limited our approach to scheduling experiments for a single service. This is acceptable as long as experiments do not involve or target more than one component or service. Otherwise, this would require additional overarching constraints that we plan to address in future work. Further, the choice and the implementation of the algorithms to identify solutions for the presented optimization problems have influence on the results. There could exist other heuristics that provide better results than the implemented algorithms. Further, there might be better ways to tailor local search and simulated annealing implementations rather than reusing the genetic algorithm's mutation operations.

Internal Validity. Threats to internal validity involve potentially missed confounding factors during result interpretation (e.g., when breaking down achieved fitness scores into the individual scores) and that the calibration of the algorithms affects the results. We mitigated this threat by performing various calibration runs with different parameter settings on 5 user groups and 25 experiments to schedule. However, as discussed earlier, calibration was a trade-off between fitness scores and execution time. Different and more fine-tuned parameters on different numbers of experiments to schedule or user groups might result in better results for the various algorithm implementations. In addition, prior work (e.g., Arcuri and Fraser [2013]) raise concerns that (hyper-)parameters of search-based techniques have strong impact on results and conclusions of studies.

External Validity. Threats to the external validity concern the generalization of our findings. Even though we used a real world traffic profile, we mimicked the distribution of users into multiple user groups which could influence our results. Further, using traffic profiles with different user interaction patterns could also lead to different results among the various algorithms. Our evaluation only relied on self-generated experiments, even though we created them based on knowledge (e.g., duration of experiments) gathered from various reports in

literature (e.g., Kevic et al. [2017], or Fabijan et al. [2017]). To mitigate this threat we created multiple scenarios (i.e., experiments with low, medium, and high required sample sizes) and evaluated the implemented algorithms on different numbers of experiments to schedule. We conducted our evaluation in a virtualized environment, i.e., Google Compute Engine. It is possible that the performance variations inherent to public clouds [Leitner and Cito, 2016] have influenced the results. To mitigate this risk, we repeated every evaluation run at least five times.

3.9 Conclusion

We formulated the problem of scheduling continuous experiments (i.e., which users participate in which experiments and when to run experiments) as an optimization problem involving three objectives. (1) experiments should not take longer than necessary to collect the required data points; (2) experiments should start as soon as possible to avoid any delay of ongoing development work; and (3), experiments should be executed on the preferred user groups to measure a certain effect. Using a weighted-sum approach we transformed these objectives into a single-objective optimization problem and we implemented a genetic algorithm, random sampling, local search, and simulated annealing to generate solutions.

Our evaluation on multiple aspects has shown that starting from 15 or more experiments to schedule, the genetic algorithm not only outperforms the other approaches when it comes to the fitness scores of the identified solutions (e.g., up to 19% for 40 experiments with high required experiment sample sizes), but also in terms of the execution time needed to find these solutions (e.g., almost a factor three for 40 experiments with high required experiment sample sizes). Currently, our approach is limited to experiments targeting a single service or component and the crossover operation does not take the validity constraints into account during reproduction. We plan to address both aspects in future work.

3.10 Acknowledgments

The research leading to these results has received funding from the Swiss National Science Foundation (SNF) under project Whiteboard (no. 149450).

4

Bifrost – Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies

Gerald Schermann, Dominik Schöni, Philipp Leitner, Harald C. Gall
Published in the Proceedings of the 17th Middleware Conference (Middleware 2016)
Contribution: prototype design, experiment design, data collection, data analysis, and paper writing

Abstract

Live testing is used in the context of continuous delivery and deployment to test changes or new features in the production environment. This includes canary releases, dark launches, A/B tests, and gradual rollouts. Oftentimes, multiple of these live testing practices need to be combined (e.g., running an A/B test after a dark launch). Manually administering such multi-phase live testing strategies is a daunting task for developers or release engineers. In this paper, we introduce a formal model for multi-phase live testing, and present Bifrost as a Node.js based prototype implementation that allows developers to define and automatically enact complex live testing strategies. We extensively evaluate

the runtime behavior of Bifrost in three rollout scenarios of a microservice-based case study application, and conclude that the performance overhead of our prototype is at or below 8 ms for most scenarios. Further, we show that more than 100 parallel strategies can be enacted even on cheap public cloud instances.

4.1 Introduction

The area of continuous delivery and deployment [Humble and Farley, 2010] is gaining more and more traction in cloud-based software engineering [Cito et al., 2015a]. Continuous delivery is a DevOps practice "intended to shorten the time between a developer committing code to a repository and the code being deployed" [Bass et al., 2015]. Shortened release cycles are essential to a company's continuing success, especially in fast-growing and contested markets such as the Web. Not only allow shorter release cycles for faster innovation, they also allow for runtime techniques to verify how users adopt new features or ideas, e.g., canary releases [Humble and Farley, 2010], A/B testing [Kohavi et al., 2013], or dark launches [Feitelson et al., 2013]. These live testing techniques share the philosophy that new versions are initially released to a small sample of the user base, and are rigorously monitored for increases in runtime faults, performance regressions [Bakshy and Frachtenberg, 2015], or changes in business metrics (e.g., conversion rate). Depending on a feature's performance, more and more users are assigned to the newer version or traffic is rerouted to previous, stable versions in order to keep the impact of malfunctioning releases low.

Unfortunately, consistently implementing live testing in large-scale applications, where new releases are deployed by many distributed teams on a daily basis, is a daunting task for release engineers. Multiple versions of software services need to be operated in parallel, and it is hard to track which runtime entity (e.g., which cloud instance or Docker container) is running which code version. A/B testing requires clear separation between versions, so as to prevent confounding factors from influencing test results. A wide range of technical and business metrics need to be constantly monitored and compared to a known

baseline for deviations. If runtime bugs, performance regressions, or unsatisfying A/B testing results are detected, a suitable fix (e.g., a rollback, or a hotfix) needs to be triggered, and if the metrics are positive, a further rollout should be considered. All of these factors make manually administering live testing often prohibitively expensive.

In this paper, we contribute to the state of the art with a formal model of live testing, which we then use as a basis for Bifrost, a prototype system for defining and automatically enacting live testing in a service-based system. Using Bifrost, release engineers can define sophisticated release strategies involving the specification of phases of canary releasing, A/B testing, dark launches, and combinations thereof, along with the associated metrics to be monitored, threshold values, and resulting actions. Release strategies are defined in a YAML-based domain-specific language [Mernik et al., 2005] (DSL), and executed via an engine implemented in Node.js. Bifrost is non-intrusive in the sense that it does not require feature toggles or other code-level changes. Instead, the middleware assumes that new releases are available as new service instances. Live testing is then implemented via traffic routing functionality.

Adopting Bifrost allows developers to formally specify how a change should be rolled out. This fosters formally or probabilistically reasoning about the strategy, e.g., in terms of expected rollout time, and enables version controlling, sharing, and reusing strategies between changes or teams. We evaluate the Bifrost approach based on a realistic microservice-based example application deployed to the Google Cloud Platform. In our experiments, Bifrost adds on average a small performance overhead of 8 ms when executing a multi-phase release strategy. This seems acceptable for many use cases, especially considering that Bifrost can be removed as soon as a change is rolled out to all users. Furthermore, our experiments show that the Bifrost middleware can support more than 100 release strategies in parallel without a significant performance degradation even when deployed to a low-end, single core cloud instance. Based on published information from industry leaders in continuous deployment, such as Facebook [Tang et al., 2015], we argue that this suggests that our approach scales to real-life release engineering scenarios.

The rest of this paper is structured as follows. Section 4.2 provides background information on live testing, and introduces a running example used in the remainder of the paper. A formal model for specifying live testing strategies is presented in Section 4.3, while the Bifrost middleware is introduced in Section 4.4. In Section 4.5, we present the results of comprehensive performance evaluation of our prototype. Finally, related previous work is covered in Section 4.6, and Section 4.7 concludes the paper by summarizing the main learnings, as well as discussing future work.

4.2 Background

In cloud-based software engineering, practices such as Dev-Ops, continuous delivery, and continuous deployment, have recently reached mainstream acceptance in the developer community. A common feature of these practices is that they provide means for software houses to further speed up their release processes and to get their products into the hands of their users faster [Schermann et al., 2016a]. For cloud-based Software-as-a-Service (SaaS) applications, this idea of "releasing faster" often comes in the form of wide-ranging automation, e.g., a deployment pipeline [Bass et al., 2015] that, fully automatedly, builds, tests, and pushes changes into production.

4.2.1 Microservice-Based Applications

As defined by Lewis and Fowler [2014], the microservice architectural style is an approach for developing a single application as a suite of small services, having each running in its own process and communicating with lightweight mechanisms, typically an HTTP resource API. Single services are independent of each other, they do not necessarily share the technology stack with other services (e.g., programming language, data storage technology). The key advantage of service-based applications is their inherent scalability and deployment options in comparison to monolithic applications. Services are scaled on a fine-granular level instead of running multiple copies of a monolithic application. Moreover,

services are deployed independently of each other, allowing replacing service versions without affecting other application parts. This architectural concept has its advantages for the adoption of live testing methods, as described in the following. It allows not only running multiple instances of a service, but also various versions of a service at the same time (e.g., canary and baseline version). Key requirement is a routing functionality ensuring that requests are correctly forwarded between the various service instances and versions. In the remainder of this paper, we will assume applications to follow this model. However, our fundamental concepts can also be implemented for other application models, for instance using feature toggles instead of dynamic traffic routing between services [Bass et al., 2015].

4.2.2 Live Testing

Moving fast in terms of releasing new features, while at the same time ensuring high quality, allows companies to take advantage of early customer feedback and faster time-to-market [Chen, 2015]. However, releasing more frequently and with a higher degree of automation also bears the risks of, occasionally, rolling out defective versions. While functional problems are usually caught in testing, performance regressions are more likely to remain undetected, as they often only surface under production workloads [Foo et al., 2015]. To mitigate these risks, SaaS providers often make use of various live testing techniques, most importantly gradual rollouts, canary releases, dark launches, and A/B testing.

Canary Releases. Canary releases [Humble and Farley, 2010] entail the concept of releasing a new feature or version to a subset of customers only, while all other users continue using the stable, previous version of the application. The idea is to test a feature on a small sample of the user base, thus testing the new version in production, but at the same time limiting the scope of problems if things go wrong. Users are either selected as a random sample of all users, based on domain-specific properties (e.g., users that ordered a specific product), or a combination thereof.

Dark Launches. Dark, or shadow, launching [Feitelson et al., 2013; Tang et al., 2015] is used to mitigate performance or reliability issues of new or

redesigned functionality when facing production-like traffic. The functionality is deployed on production environments without being visible or activated for any end users. However, some or all production traffic is duplicated and applied to the "shadow" version as well. This allows the provider to observe how the new feature would be behaving in production, without impacting any users,

Gradual Rollouts. Gradual rollouts [Humble and Farley, 2010] are often combined with other live testing practices, such as canary releases or dark launches. The amount of users testing the newest feature or functionality is gradually increased (e.g., increase traffic to the new version in 5% steps) until the previous version is completely replaced.

A/B Testing. A/B testing [Kohavi et al., 2013] is technically similar to the other live testing techniques discussed here, but is mainly used for differing goals. While all the techniques so far are used to evaluate a new version with regard to a baseline (the presumably stable, previous version), A/B testing is often used to compare two new, alternative, implementations of the same functional requirement. These two versions are run in parallel, with 50% of all requests going to either version. Whereas it is common to select users with particular features for canary releases, A/B tests usually require a uniform sampling of the entire user demography for both alternatives. After a predefined experiment time, metrics (e.g., conversion rate) are statistically evaluated to decide which version fared better (or whether there was a statistically significant difference at all).

4.2.3 Example Live Testing Strategy

A core observation underlying this paper is that rollouts in practice often consist of multiple sequential phases of live testing. For instance, a concrete rollout strategy may consist of initial dark launching, followed, if successful, by a gradual rollout over a defined period of time. If no problems, are discovered, the new change may be A/B tested against an alternative implementation, which may have run through a similar live testing sequence.

A simple example live testing strategy, which will be used throughout the remainder of the paper as a running example, is given in Figure 4.1. Assume a

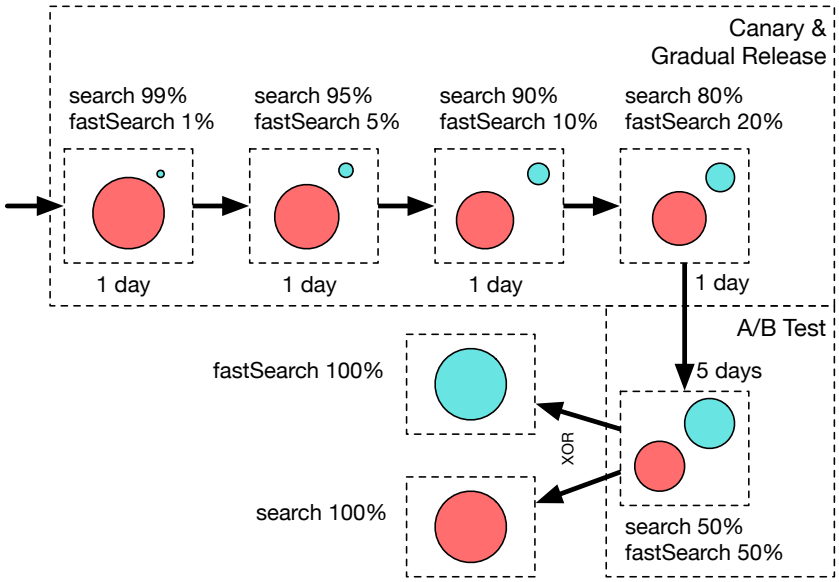

Figure 4.1: A simplified example of a live testing strategy with multiple phases. A change is gradually rolled out to more and more users, and subsequentially A/B tested.

company hosting a service-based web application selling consumer electronics. One of the integral services is the *search* service allowing customers to look for products they are interested in and to get an overview of the product catalog. The search service shall be redesigned and implement a new algorithm for delivering more accurate search results based on other users' search requests and their buying patterns. As replacing the previous slow, but working, search service by the new one is associated with risks, the service shall be canary tested first. Once the service performs as expected from a technical perspective, an A/B test should be conducted between the stable and canary variant. In case that the new implementation performs better according to a priori defined business metrics, a complete rollout should happen, otherwise a fallback to the stable version is conducted. The canary tested reimplementation *fastSearch* shall be rolled out to 1% of the US users first. Search and fastSearch are continuously monitored and collected metrics include response time, processing time (i.e., how long does the actual search algorithm take to get results), number of 404 requests,

and the number of search requests per hour. Thresholds for fastSearch are set based on historic values collected for the stable search service, e.g., response time below 150ms. On a daily basis, and as long as the monitored metrics do not show any abnormalities, fastSearch shall be gradually rolled out to more and more users, first to 5%, then 10%, 20%, until at 50% the A/B test is conducted as shown in Figure 4.1. Besides more technical metrics, the A/B test focuses also on a business perspective (e.g., comparing the number of sold items on both variants), and is conducted for 5 days to capture enough data supporting statistical reasoning. State-of-the-art tools, such as the Configurator used by Facebook [Tang et al., 2015], require strategies such as this running example to be manually implemented by a human release engineer, for analyzing the data and the tweaking the rollout configuration after every step. This is labor-intensive, error-prone, and requires substantial experience in data science. In this paper, we propose Bifrost as an automated and more principled approach towards managing such release strategies.

4.3 A Model of Live Testing

Before explaining the implementation of the Bifrost middleware, we first introduce the fundamental ideas and characteristics that the system is based on, as well as the underlying formal model.

4.3.1 Basic Characteristics

After thorough analysis of live testing in general, and the practices discussed in Section 4.2 specifically, we have identified the following basic characteristics of a formal model for live testing.

Data-Driven. Live testing require extensive monitoring to decide on test outcomes or evaluate the current health state. This monitoring data is collected using existing tools in the application's landscape using Application Performance Monitoring, such as Kieker [van Hoorn et al., 2012] or New Relic[1]. A model of

[1] `https://newrelic.com`

live testing needs to support the inclusion of monitoring data into its runtime decision process.

Timed Execution. Live testing requires the collection, analysis, and processing of data in defined intervals. Gradual rollouts depend on timed increments to gradually introduce new versions or control the routed traffic. Depending on the concrete usage scenario, these methods may stretch over minutes, hours, or days.

Parallel Execution and Traffic Routing. All live testing practices require the parallel operation of multiple versions of a service, e.g., a stable previous version and an experimental new implementation for canary releases, or two alternative implementations for A/B testing. This also requires the correct routing of users to a specific version. For instance, canary releases are often targeted at specific user groups. For A/B tests, it is often important that the same user is directed to the same implementation across sessions.

Ordered Execution. Ordered execution is required to form live testing strategies consisting of chained phases of canary releasing, gradual rollouts, dark launches, and A/B tests. An example for such a live testing chain is given in Section 4.2.

4.3.2 Live Testing Model

Based on these identified characteristics, we derived a formal representation for live testing strategies. To begin with, a *strategy* $\mathcal{S}$ is modeled as a 2-tuple:

$$\mathcal{S} : \langle \mathcal{B}, \mathcal{A} \rangle$$

A strategy $\mathcal{S}$ consists of a set of *services* $\{b_1, \ldots, b_n\}$ and a deterministic finite automaton $\mathcal{A}$. In our model, services $b_i \in \mathcal{B}$ represent atomic architectural components, for instance services in a microservice-based system. Services b_i themselves are available in different versions (e.g., a stable previous search service version, and an experimental new version) or as alternative implementations (e.g., for A/B testing). Whenever a change is rolled out, a new service version is launched. For a service b_i, this is modeled as a tuple $\langle v_1, \ldots, v_n \rangle$. Moreover, each

of those versions v_i is associated with static configuration information sc_i, which holds a version's endpoint information (e.g., host name, IP address, and port). A user $u_i \in \mathcal{U}$ connected to the system is always using exactly one version of a service. However, this assignment may change during the execution of a release strategy (e.g., during a gradual rollout a user may be reassigned from a stable version to the canary version). Thus, this dynamic routing information, i.e., to which version v_j of a service b_i a user u_k is assigned to, modeled as a 3-tuple $\langle u_k, v_j, sticky \rangle$, represents an important part of a service's routing state. *Sticky* is a boolean flag specifying if a user's assignment is permanent for a certain state, thus whether a subsequent request by a user (e.g., search request) may be routed to a different version or not. Dark launches are different from all other live testing practices, in that they duplicate rather than reroute a traffic to a specific service version. This is modeled as a 3-tuple $\langle v_{i,j}, v_{k,l}, p \rangle$, where $v_{i,j}$ denotes the source version from which p percent of the traffic is duplicated and also routed to the target version $v_{k,l}$. Thus, the dynamic routing configuration dc_i of a service b_i is a 2-tuple $\langle \mathcal{M}, \Gamma \rangle$ being $\mathcal{M}$ a tuple of user mappings $\langle u_k, v_j, sticky \rangle$ and Γ a tuple of dark launch routing information $\langle v_{i,j}, v_{k,l}, p \rangle$.

The execution state of a release process is represented by an *automaton* $\mathcal{A}$, which is defined by a 5-tuple $\langle \Omega, S, s_1, \delta, F \rangle$. Ω represents the monitoring data a live testing strategy uses for decision making. Ω is modeled as tuple of metrics $\langle m_1, \ldots, m_n \rangle$, each m_i representing a time series $(t_0, \ldots, t_n)$ of metric values over time (t_0 to t_n). This data typically originates from external monitoring solutions. The automaton itself is defined as a set of states $\{s_1, \ldots, s_n\}$, $s_i \in S$, a starting state s_1, and set of final states F, where $F \subseteq S$. δ is a state transition function specifying the subsequent state depending on the current state and the outcome of a state's associated checks ($e \in \mathbb{Z}$), formally defined as $\delta : S \times \mathbb{Z} \to S$. States and transitions represent the concept of ordered execution, in which multiple states form distinctive phases during the live testing process.

A state s_i is defined as a 5-tuple $\langle \mathcal{C}, \mathcal{T}, \mathcal{W}, \Phi, \eta \rangle$ including checks $\mathcal{C}$, thresholds $\mathcal{T}$, weights $\mathcal{W}$, configurations Φ, and a user selection function η. In a state, multiple checks modeled as a tuple $\langle c_1, \ldots, c_n \rangle$, $c_i \in \mathcal{C}$, are executed at the same time, thus matching the characteristic of parallel execution. A check may for

instance represent monitoring a specific metric (e.g., service response time). The outcomes of each individual check are combined as a weighted linear combination. The resulting outcome value serves as input for the state transition function δ. For each check c_i, there is a weighting factor $w_i \in \mathcal{W}$, thus formally, weights are modeled as a tuple $\langle w_1, \ldots, w_n \rangle$.

Each state is associated with specific services' dynamic routing configurations Φ, modeled as a tuple $\langle dc_j, \ldots, dc_k \rangle$ containing all configurations of services relevant for a state. The user mappings $\mathcal{M}$ of those dynamic routing configurations are built and controlled by the state's function η, formally $\eta : \mathcal{U} \rightarrow \mathcal{V}$, which assigns a specific user u_i to a version v_j of service s_k. This allows fine-grained routing and filtering functionality, e.g., assign 5% of US users to the *fastSearch* canary. Our approach is agnostic to how this selection and filtering is implemented. For instance, our approach is compatible to the user selection and sampling approach used in Configurator [Tang et al., 2015]. Once the execution of a release strategy enters a state s_i, the dynamic routing configurations of the services associated to this state are evaluated and executed.

An example state machine for the running example introduced in Section 4.2 is given in Figure 4.2. In state b, the stable *search* service is assigned to 95% of the users, while the *canary tested*, newly designed reimplementation *fastSearch* is used by 5% of the users. Depending on the outcome of the various checks in each state and their weighting, a numerical outcome value is generated in each state. This outcome value is compared against defined *thresholds*, leading to a state transition. For instance, in state b, a transition either happens directly to state d because of the canary's good performance (outcome > 4), to state c in which the traffic is only slowly increased (outcome $= 4$), or a rollback happens transitioning to state g (outcome mapped ≤ 3).

Formally, the state transition function δ takes the current state s_i and the state's aggregated and weighted outcome value $e \in \mathbb{Z}$ as input. For each state, an ordered tuple of thresholds $\langle t_1, ..., t_n \rangle$, $t_i \in \mathcal{T}$, is specified containing at least one value. A tuple of thresholds with n values forms $n + 1$ disjoint ranges, e.g., thresholds $\langle 2, 4 \rangle$ form the ranges $-\infty < x \leq 2$, $2 < x \leq 4$, and $4 < x \leq \infty$. In the state transition function δ, to each range of a state s_i, a state s_j is assigned,

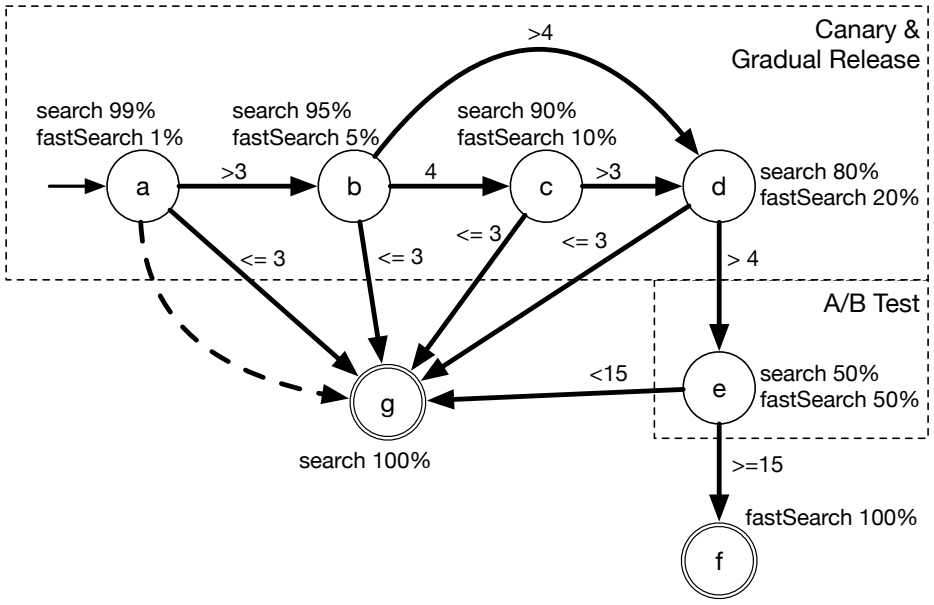

Figure 4.2: A visualization of the state machine of the running example. Every state executes checks for a specified amount of time, leading to a numerical outcome value. State transitions are based on this outcome. State "g" represents a rollback of the release. The dashed arrow in state "a" represents an "exception" that allows to jump directly to the rollback state "g" if a serious problem is detected in "a".

representing the automaton's subsequent state if the aggregated outcome value e falls into the range. In Figure 4.2, state a has exactly one threshold (i.e., 3), thus forming exactly two possible state outcomes, while state b has two thresholds (i.e., $\langle 3, 4 \rangle$) and thus three outgoing transitions.

In the following, we will elaborate step by step how a state's outcome value is determined and when state transitions are triggered. A single state executes multiple checks at the same time. A check c_i is defined as a 3-tuple $\langle f_{c_i}, \Omega^i, \tau \rangle$ consisting of a metric evaluating function $f_{c_i} : \Omega^i \to \{0, 1\}$, monitoring data Ω^i, and a timer τ.

In detail, a check c_i's function f_{c_i} takes a subset of the monitoring data $\Omega^i \subseteq \Omega$ as input and returns 0 or 1, i.e., a check is either successful or not. Such evaluation functions could be of varying complexity, they might check only for a single

service version's metric (e.g., response time < 150ms), a combination of multiple metrics of a version, or evaluate even metrics across multiple services and versions (e.g., for the purpose of A/B testing).

In order to reason about a service's behavior, it is necessary to continuously evaluate the adherence to specified metrics, thus a check's evaluation function may be executed multiple times. This is achieved by introducing a timer mechanism τ, controlling when and how often single checks execute. Functions evaluating monitoring data are executed independently of each other. This is illustrated in Figure 4.3 showcasing the timed (re-)execution of the functions associated with three checks using different execution intervals. As the outcome of a single function execution is either 0 or 1, the outcome of a check is determined by aggregating (i.e., summing up) the outcome values of each execution (i.e., 1 to n) during the course of time controlled by τ leading to an outcome value $e \in \mathbb{Z}$.

$$\begin{aligned} f_{c_i}^{\tau}(\Omega^i) : f_{c_i}^{1}(\Omega^i) + \ldots + f_{c_i}^{n}(\Omega^i) \\ = \sum_{j=1}^{n} f_{c_i}^{j}(\Omega^i) \rightarrow e \in \mathbb{Z} \end{aligned}$$

The model distinguishes between two types of checks: *basic* checks and *exception* checks. While for basic checks the single execution results are only evaluated at the end, single execution results of exception checks trigger state transitions whenever their evaluation function returns 0. The intuition here is that for basic checks, individual tests may fail (e.g., even if a change performs as expected, there may be a small number of individual checks for a change for which the error rate slightly increased due to expected stochastic variations). However, if things are going very badly (e.g., a 100% or higher increase in the error rate), exception checks allow developers to immediately roll back a release without having to wait to the end of the current state. In Figure 4.3, such state changes could happen at t_0, t_1, t_2, and t_3. State a in Figure 4.2 contains an exception check leading to state g.

Formally, an exception check c_i is a 4-tuple $\langle f_{c_i}, \Omega^i, \tau, s_j \rangle$ consisting of a metric evaluating function f_{c_i}, monitoring data Ω^i, a timer τ, and a fallback

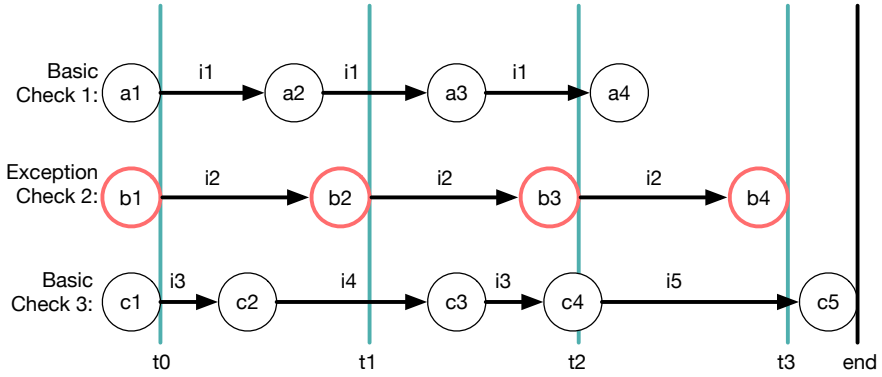

Figure 4.3: An illustration of the time-based execution of multiple checks.

state $s_j \in S$ to which the automaton switches if the evaluating function returns 0 during its timed (re-)execution. If all n function executions are successful, the aggregated outcome value of an exception check equals n.

A basic check c_i is a 5-tuple $\langle f_{c_i}, \Omega^i, \tau, T_{c_i}, Out_{c_i} \rangle$ of a metric evaluating function f_{c_i}, monitoring data Ω^i, a timer τ, an ordered tuple of thresholds $\langle t_1, ..., t_n \rangle$, $t_i \in T_{c_i}$, and an output mapping Out_{c_i}.

$$Out_{c_i} : \{(t_i, t_{i+1}, r_i) \mid t_i < t_{i+1}, r \in \mathbb{Z}, t \in T_{c_i}, i \in \{1, \ldots, n\}\}$$

Similar to a state's outcome in the state transition function δ, the aggregated outcome of a basic check e is compared to thresholds forming disjoint ranges, and based on the check's outcome mappings Out_{c_i}, mapped to an integer value r_i.

$$e \xrightarrow{Out_{c_i}} \{r_i \mid t_{i-1} < e \leq t_i, (t_{i-1}, t_i, r_i) \in Out_{c_i}\}$$

Thresholds are used to cope with varying monitoring data, e.g., the response time of the monitored *fastSearch* service may vary, thus the outcomes of the evaluation function may vary as well. Outcome mappings allow mapping those different outcome values onto a normalized integer outcome value. For example, assume a basic check for controlling *fastSearch*'s response time in state b in

Figure 4.2. The check is executed 100 times in intervals of 10 minutes. The response time check's thresholds are 75 and 95, thus forming ranges $x \leq 75$, $75 < x \leq 95$, and $x > 95$. The corresponding mappings are $(-\infty, 75, -5)$, $(75, 95, 4)$, and $(95, \infty, 5)$. This means that if the check fails more than 24 times, the mapping returns -5, if the aggregated value is between 75 and 95, it returns 4, otherwise 5.

Once we have the results of the single checks of a state, the final step is to aggregate those results as a weighted linear combination, and consider their weighting factors in order to determine the state's outcome.

$$\sum_{i=1}^{n} f_{c_i}^{\tau}(\Omega^i) * w_i \rightarrow e \in \mathbb{Z}$$

Given the current state s_i, this final result e is the input for the state transition function δ, resulting in either a state change, or staying in the current state. In this case, the state is re-executed, with all timers and thresholds reset. This concept of multiple outgoing paths allows (1) continuing the rollout strategy if the tested services behave as expected, (2) staying in a certain state if results are not definite and require reexecution, or (3) switching to a fallback state if new functionality does not behave as expected and to keep its impact low. Moreover, the concept of exception checks allow state changes (i.e., roll backs) at any time during the execution.

4.4 Bifrost

In this section, the Bifrost middleware is presented. The system is a Node.js based prototype implementation of our live testing model. Our prototype specifically targets micro-service-based applications.

4.4.1 System Overview

As visualized in Figure 4.4, the two main components of the Bifrost middleware are the Bifrost engine and Bifrost proxies. The middleware acts on top of

the application's services, ensuring that routing instrumentation specified in the release strategy is adhered to.

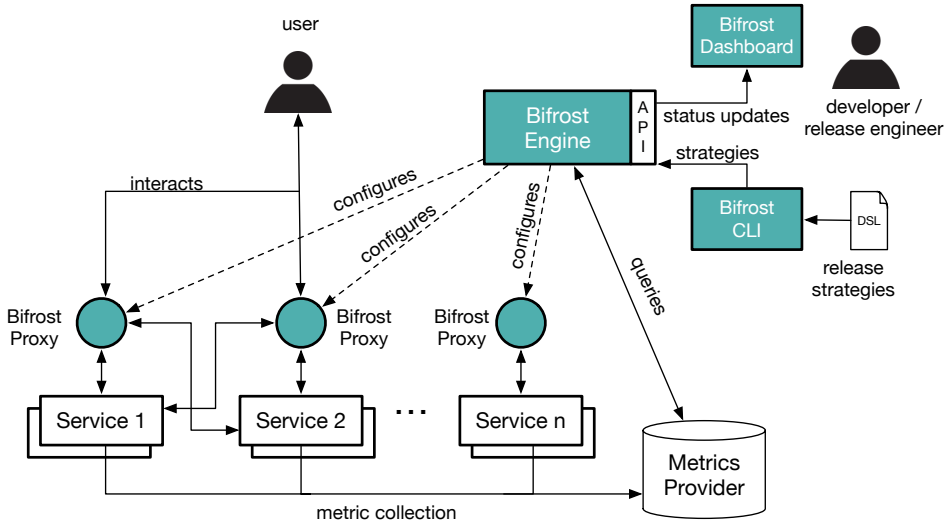

Figure 4.4: High-level architectural overview of the BIFROST middleware.

Conceptually, there is exactly one BIFROST proxy for each service that is part of the applied live testing method. This one-proxy-per-service concept prevents traffic bottlenecks and keeps services decoupled. A service acting behind by a proxy may run in multiple instances and multiple versions at the same time. BIFROST proxies facilitate live testing via implementing dynamic traffic routing. For instance in case of an A/B test, 50% of all traffic is routed transparently to two different versions of a service. A key advantage of this design is that the middleware is easy to integrate into existing applications, without altering or rewriting functionality. Thus routing and rollout logic is not part of the services' code bases, as would be the case for feature toggles [Bass et al., 2015]. The middleware supports any web-based service including databases and external services accessed through HTTP. BIFROST proxies are lightweight. Each instance of the proxy is basically another service added to the application, and proxies work in combination with load balancers, auto-scaling functionality, reverse proxies or request gateways. The BIFROST engine has the main responsibility to

orchestrate and properly configure the deployed proxies in the system. Basically, the engine executes the state machine of the formal release model. It interprets the release strategies specified in a domain-specific language, and continuously queries and observes monitoring data collected by metrics providers or external services in order to evaluate the rules specified in the release strategies and enact appropriate actions (i.e., state changes). Whenever a state change happens during the rollout process (e.g., entering a new phase in the specified strategy), the engine updates the affected proxies.

Besides the middleware components, Bifrost comprises two additional tools, the Bifrost command-line interface (CLI) and Bifrost dashboard. The CLI connects to the Bifrost engine and allows scheduling and executing release strategies remotely or as part of release scripts (e.g., build automation using Jenkins). The Bifrost dashboard visualizes the current execution state of release strategies providing detailed information such as the outcome of executed checks (e.g., metric below threshold).

4.4.2 Implementation

We now discuss how this high-level design has been realized in Bifrost.

Technology Stack

The Bifrost middleware has been developed mainly in JavaScript utilizing Node.js as the server-side JavaScript runtime, in combination with Babel[2], which is a backwards-compatible JavaScript transpiler. Node.js was chosen due to its lightweight and efficient architecture that favors event-driven applications, which Bifrost heavily uses due to the asynchronous nature of release process (e.g., checks running in parallel with different timer configurations). The communication between the middleware's components is handled through RESTful HTTP APIs that make use of ExpressJS[3]. Moreover, Socket.IO is used implementing the WebSocket protocol providing full-duplex communication channels. This is nec-

[2] `https://babeljs.io/`
[3] `http://expressjs.com/`

essary for updating the Bifrost CLI and dashboard with real-time information. Finally, the proxy functionality has been implemented using node-http-proxy[4].

Domain-Specific Language

To simplify the specification of release strategies and thus to avoid specifying every single state of the underlying formal model, the Bifrost domain-specific language (DSL) was designed. Besides fostering simplicity, the text-based DSL aims to be version-controlled, thus supporting transparency and traceability of a company's release strategies. The DSL was built as an internal DSL on top of YAML as a host language. YAML is a data serialization language designed to be readable by humans. In the following, we will present implementation details of and design decisions for the engine based on small DSL code snippets showcasing specific elements of a rollout strategy. However, a more detailed description of the DSL is out of the scope of this work, but example strategies formalized in the DSL that have been used throughout the evaluation of Bifrost are part of our online appendix.

Data-Driven Execution. Collected and aggregated monitoring data is the essential ingredient for the engine's runtime decisions. The Bifrost engine is designed to support multiple data sources. However, currently, the engine's prototype implementation is primarily built for Prometheus [2019]. Listing 4.1 shows an example how a *basic check* is implemented in the Bifrost DSL in form of a *metric* element.

Lines 2 to 6 specify the data retrieval, i.e., to which provider to connect to and which query to be executed. The metric providers' access information (i.e., IP, port) is specified in a configuration file loaded at the engine's start-up. In this concrete example, the query retrieves the amount of request errors associated with the service instance *search* from Prometheus. Bifrost supports retrieving an arbitrary number of metrics from different data providers in the context of a check. The retrieved data is then associated to the provided name and can be used inside the scope of the check for validation purposes.

[4] `https://github.com/nodejitsu/node-http-proxy`

```
1  - metric:
2    providers:
3      - prometheus:
4        name: search_error
5        query: request_errors
6                 {instance="search:80"}
7    intervalTime: 5
8    intervalLimit: 12
9    threshold: 12
10   validator: "<5"
```

Listing 4.1: Example Metric

Timed Execution. Each basic check in our model has a metric evaluating function, which operates on a set of metrics, and its execution is controlled by a timer. In the previous step, we have already shown how the engine collects metrics. Line 10 of Listing 4.1 shows a simple function evaluating the collected metrics. In this case, a single metric is retrieved and compared to a scalar value. The check is reexecuted every 5 seconds and 12 times in total. The current implementation of the DSL represents a simplified version of the release model discussed in Section 4.3.2. Each check has exactly one threshold value, thus the aggregation of the result of a check's timed-execution can be mapped to either true or false. In line 9, the threshold is set to 12, which means that the check returns only true if all 12 executions evaluate to true.

Rollouts. The parallel execution of checks and their aggregated outcomes may lead to state changes, which then influence how traffic is routed through the system, thus changing dynamic routing configurations dc_i of services b_i. The basic instrument for specifying such rollouts is the *route* directive in the Bifrost DSL. An example for a route supporting dark launches is provided in Listing 4.2.

The example specifies that all traffic (line 6) routed to the *search* service within the next 60 seconds (line 8) shall be duplicated (line 7) and also routed to the *fastSearch* service. This allows dark launching a service, thus assessing amongst others whether the tested service scales correctly.

```
1 - route:
2   from: search
3   to: fastSearch
4   filters:
5     - traffic:
6       percentage: 100
7       shadow: true
8   intervalTime: 60
```

Listing 4.2: Dark Launch

In order to support such mechanisms, Bifrost proxies intercept incoming connections, and depending on their configuration, they route requests accordingly. Bifrost supports two types of routing: header-based and cookie-based. The former inspects a request's header fields (specified in RFC 2616), which could include custom-named header fields as well. For header-based traffic filtering, the proxy itself does not decide to which service instance a request is routed, it acts solely on its configuration received from the engine. Thus, the concrete header field has to be injected somewhere else in the process, e.g., by an external service called at the user's login controlling which users are in which group of a conducted A/B test. This is different for the second option, cookie-based filtering, where, for example in case of A/B tests, the proxy decides into which bucket a request is put into. Listing 4.2 shows an example for such a cookie-based filtering variant. In addition, this concept is used for applying general random traffic filtering such that a certain percentage of users is assigned to a specific version. However, depending on the type of the conducted release practice, it may be important that requests from the same users are always routed to the same service instance (e.g., A/B testing). This behavior is generally called sticky sessions. The proxy accomplishes this by setting a cookie on the client using the *Set-Cookie Header* in its response. The cookie contains a RFC-compliant UUID that is used to re-identify the client in subsequent requests. Depending on whether sticky sessions are used or not, the proxy either stores the set cookie to re-identify users, or the subsequent request is again running through the proxy's decision process.

Deployment Configuration. Evidently, the engine needs to be aware of which services exist in the system, and where the proxies are located. This corresponds to the static routing information modeled in the formal release model. In the Bifrost DSL, this is covered by the DSL's deployment part, while the specification of the previous code snippets where all in the DSL's strategy part. The former takes a list of key-value pairs mapping host names of services to host names of corresponding Bifrost proxy instances. This simple mechanism allows the tool to work in different deployment setups. The middleware per se is not responsible for the deployment of the various components. However the DSL and engine are designed in such a way to be extended and make use of deployment management tools, such as Chef or Puppet, in future versions.

4.5 Evaluation

The Bifrost toolkit provides developers with a flexible approach to introduce various rollout practices into their release process. However, the feasibility of this approach is influenced by the middleware's performance impact and how well the approach scales, both conceptually and technically. Thus, in the following section we specifically take a look on how the Bifrost middleware performs in realistic settings. We look at two different scenarios, evaluating the performance overhead introduced by the Bifrost proxies for the end user as well as the scalability of the Bifrost middleware itself, in terms of parallel strategies and checks. A replication package for our study is available in the online appendix.

4.5.1 Evaluation of End-User Overhead

We firstly address the question whether using Bifrost degrades end user performance.

Case Study Application

To address this question, a case study application simulating a generic microservices application was necessary. Unfortunately, few suitable open source

microservice-based applications exist. Hence, we developed a custom Node.js based case study application specifically to run performance tests against for the purpose of evaluating the middleware. The implementation of this case study is available in the online appendix.

This application simulates a generic e-commerce website selling consumer electronics. It was kept simple in order to provide a testbed for the performance evaluation and demonstration of the capabilities of the Bifrost middleware. The application consists of 7 services in total: a HTML/JavaScript *frontend*, and three RESTful HTTP services, *product*, *search*, and *auth*. The *product* service allows browsing the product catalog and placing buy orders, the *search* service is used for executing text-based product search queries, and *auth* service authenticates and authorizes users based on their provided e-mail and password, and validates tokens. In addition, there is a *MongoDB* database for storing products and users, an instance of *Prometheus*, which collects container and low-level performance metrics as well as business metrics from services that expose them, and finally *nginx*[5]. Nginx is a reverse-proxy used as a central entry-point to the application for users. It proxies incoming requests to either the frontend service or to the product service. An overview of the case study application architecture is provided in Figure 4.5. Connections between the services and Prometheus were omitted for clarity reasons.

Experiment Setup

We now discuss how we have set up the case study application and experiment.

Case Study Application Deployment. We deployed the case study application on 12 virtual machines forming a Docker Swarm[6] on the Google Cloud Platform[7]. We used virtual machines of type `n1-standard-1` in Google's `us-central1-a` region. Consequently, each virtual machine had a single virtual CPU implemented as a single hardware hyper-thread on a 2.6 GHz Intel Xeon

[5] `https://www.nginx.com`
[6] `https://docs.docker.com/swarm/`
[7] `https://cloud.google.com`

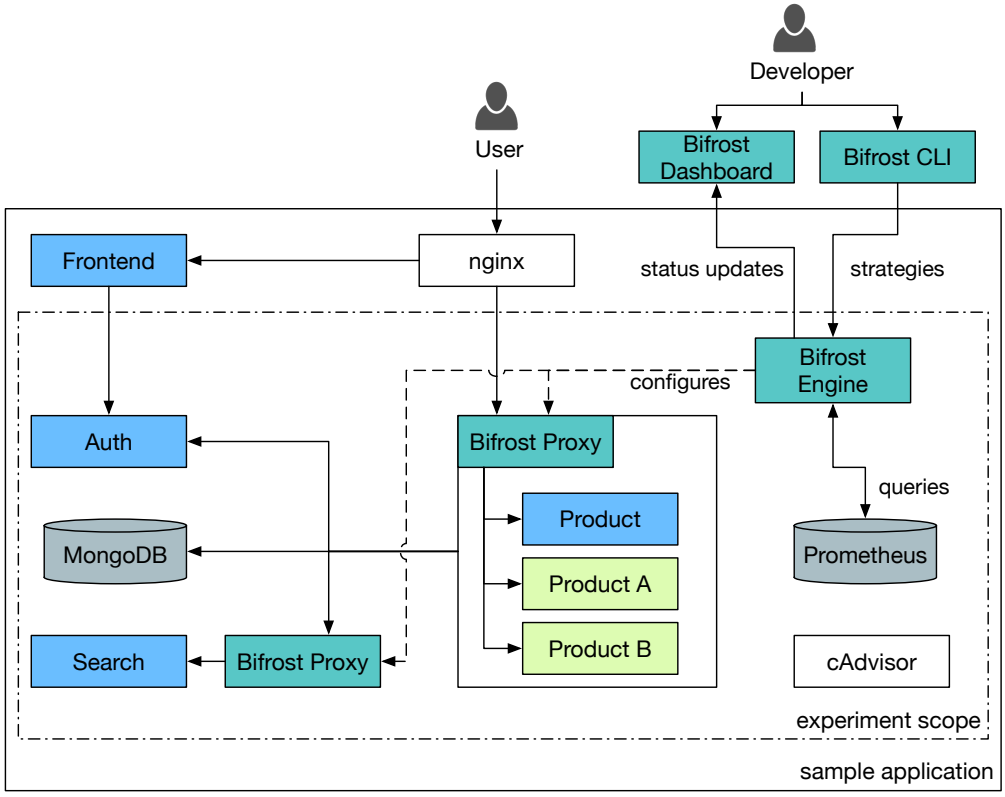

Figure 4.5: Architecture of a microservice-based case study application, consisting of 7 microservices.

E5 and 3.75 GB memory. Experiments were conducted between May 1st and May 19th, 2016.

The first node acted as Swarm-Master. Docker Swarm allows clustering a pool of Docker hosts into a single virtual Docker host supporting the execution of Docker Compose, which simplifies application deployment and in our case replication as well. Every service of the case study application resides in its own Docker container. Moreover, to ensure that a single container's performance does not influence other containers, in this setup, all containers were running on their own virtual machine. Besides the services of our case study application, the middleware components were deployed as Docker containers as well, i.e., one VM hosting the Bifrost engine, and two VMs hosting proxies for the `search` and

`product` service. In addition, to automate the evaluation process, the Bifrost CLI was put into a dedicated container as well. As the `auth` service is not relevant for the executed live testing strategy, it does not use a Bifrost proxy. This simulates the case of a stable service for which currently no live testing strategy is executed. To collect the containers' performance metrics (e.g., CPU utilization, memory consumption) cAdvisor[8] was used pushing the collected data to Prometheus, which further increased the number of containers and VMs in our experiment setup by two. Finally, to simulate production traffic, we used another Docker container and VM of the same type for hosting an instance of Apache JMeter as load generator.

Test Setup. The goal of this experiment was to show the performance impact of the Bifrost middleware in a more complex release cycle consisting of the execution of a release strategy involving multiple live testing methods. In this scenario, the product service shall be replaced and two new alternatives were implemented for this purpose, `product A` and `product B`. The specified release strategy introduces both alternatives to the running system, runs a set of live testing methods making sure that they perform as expected and depending on the outcome of those tests, one of the newly implemented product services shall be gradually rolled out to all users. The release strategy involves the following phases.

1. Canary Launch: Tests `product A` and `product B` service while monitoring for errors, i.e., HTTP status code 500 responses. 5% of the traffic to the stable product service gets redirected to A and B respectively, and an aggregated error count from Prometheus is monitored. This phase lasts for 60 seconds, and is implemented using cookie-based routing without sticky sessions. This phase corresponds to a single state in the formal model with two checks running in parallel, which are re-executed every 12 seconds.
2. Dark Launch: `Product A` and `product B` receive 100% of all original traffic to the product service for a duration of 60 seconds. This represents a single state in the formal model. We refrained from our initial checks on

[8] `https://github.com/google/cadvisor`

the services' CPU utilization as this would have led, in certain cases, to automatic rollbacks during our load test.

3. A/B Test: Routes 50% of the product traffic to `product A` and the remaining 50% to `product B`. As a test metric the sales performance is monitored over 60 seconds. The test uses sticky sessions and cookie-based routing. After completion, the traffic distribution is reverted to the original product service. This live test corresponds to a single state in the model, with one check executed at the end.
4. Gradual Rollout: Rolls out the winner from the previous A/B Test starting with 5% traffic up to 100%, increasing traffic 5% every 10 seconds, for 200 seconds duration in total. Corresponds to 20 states in the model.

Note that, in order to compress the total duration of the experiment to 380 seconds, we chose extremely short execution times for each phase. Obviously, in practice, developers would typically choose longer durations for each phase.

We initiated the execution of the live testing strategy after a ramp up period of 30 seconds to slowly increase the load and after an additional 60 seconds for health checking the deployed services. After the ramp up, a steady traffic of 35 requests per second was simulated using a JMeter test suite. The test suite targeted the product service and consisted of 4 different requests that touched different parts of the system:

- Buy: A HTTP POST request to the `product` service, which writes to the database. No response body is sent back.
- Details: A HTTP GET request to the `product` service, which returns information about a single product. The request only requires a read operation in the database, and returns a small response body.
- Products: A HTTP GET request to the `product` service, returning a list of all products including their buyers. Requires a read operation in the database as well, but returns a large response body.
- Search: A HTTP GET request to the `product` service, which in turn invokes the `search service`. Requires another read operation in the database, and returns a small response body.

All requests require authorization via the `auth service`. We conducted test runs in three different variations: (1) baseline, i.e., running the load test without the middleware and proxies deployed, (2) Bifrost inactive, running the load test with the middleware and proxies deployed but without executing any strategy, and (3) Bifrost active, running the load test with the middleware and proxies deployed and executing a strategy. For each of those three variations we collected the average response time in 5 test runs and used a moving average with a window size of 3 seconds for aggregation.

Test Results. Figure 4.6 plots the average end user response time as measured by the JMeter load generator during the release in the described phases (canary launch, dark launch, A/B test, gradual rollout). The single release phases are highlighted for better readability. We observe that, in general, Bifrost introduces a constant small overhead to service invocations. For gradual releases and canary tests, this overhead is approximately 8 ms in our tests (see also Table 4.1 for detailed numbers), which we consider acceptable for many production settings. Further, it should be noted that our Node.js based prototype implementation is not optimized for speed, and a more efficient implementation would likely be feasible. Further, our evaluation setup made use of cookie-based routing, which is generally slower than a header-based routing would be. Finally, this case study application and all components have been deployed on low-end cloud instance types. More powerful instance types, or dedicated server hardware, would likely reduce the overhead further. However, even with this prototype implementation we have shown that our underlying concept seems feasible for real-world usage. Another observation from Figure 4.6 is that response times are stable within phases. That is, there is no middleware-induced change in the overhead during tests, which is particularly important for A/B testing. The scenarios when Bifrost is inactive and active did not lead to statistically significant response times for canary releases and gradual rollouts, indicating that the execution of a single strategy is cheap. This will be researched in more detail in Section 4.5.2.

Two phases need more explanation, specifically the A/B test and the dark launch. For the A/B test (third phase in the figure), we observe that the average

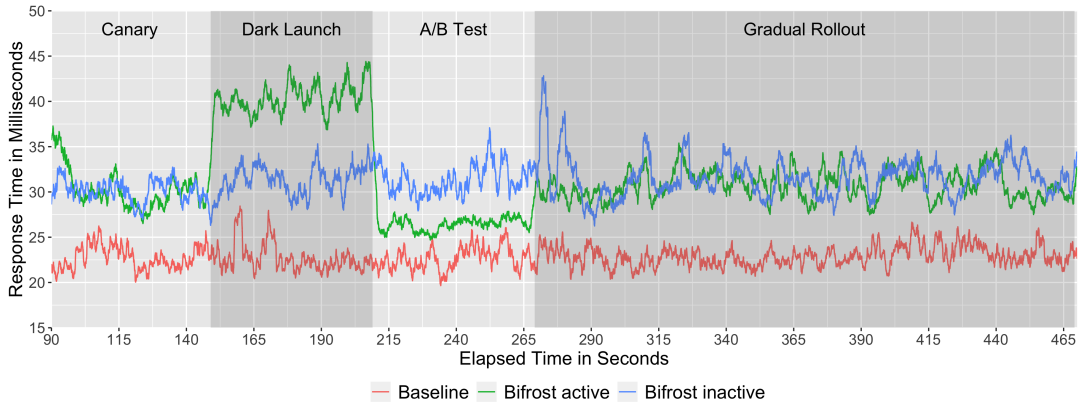

Figure 4.6: 3-second moving average of response times as monitored in the JMeter load generator over the duration of the experiment. Baseline is the response time without Bifrost, inactive represents Bifrost, and specifically the routing proxies being installed but without any active strategy, and active is the case when a live testing strategy is being executed.

	Canary			**Dark Launch**			**A/B Test**			**Gradual Rollout**		
	baseline	inact.	active	baseline	inact.	active	baseline	inact.	active	baseline	inact.	active
mean	22.75	30.04	30.28	22.68	31.34	40.23	22.64	31.30	26.52	22.93	31.59	30.68
min	20.04	26.27	26.47	20.42	27.95	31.67	19.65	27.86	24.65	20.35	26.20	27.43
max	26.24	32.79	38.58	28.44	35.26	44.35	26.05	37.03	31.67	26.66	42.76	35.34
sd	1.26	1.21	2.22	1.53	1.53	1.70	1.23	1.58	1.00	1.07	2.18	1.55
median	22.58	30.08	29.77	22.36	31.51	40.11	22.59	30.98	26.49	22.85	31.43	30.53

Table 4.1: Basic statistics of response times in milliseconds for all release phases.

response time decreases in comparison to when Bifrost is inactive. This is a side-effect of the load balancing effect of A/B testing, i.e., in this phase invocations are by definition split between two services, leading to reduced load on both of them. In effect, this reduces the overhead to approximately 4 ms. For the dark launch, we observed the opposite effect. As this live testing strategy requires duplication of traffic, the overhead induced by the middleware is increased as well, leading to an overall higher response time and an increased overhead of 18 ms. This is because in our test setting three requests need to be shadowed (requests to the authentication service, the product service, and the database). Thus, in contrast to other live testing methods, dark launching requires a certain

level of caution (e.g., making sure that the proxy runs on machine able to handle the load), especially if, as in our setup, 100% of the traffic is duplicated.

4.5.2 Evaluation of Engine Performance

The previous performance test focused on the overall application's performance. However, as we executed only a single release strategy, we now want to study how the Bifrost middleware behaves under load created by (1) executing multiple release strategies at the same time (simulating the case of a large organization with many teams, all independently releasing new versions), and (2) executing complex release strategies with an increasing amount of parallel checks.

Executing Multiple Release Strategies

This test studies how many parallel live experiments can be conducted at the same time, and, thus, whether our middleware is capable of being used in a broader context in a company having various different product teams launching rollout experiments independently from each other.

Case Study Application Deployment. We used a cluster of 4 virtual machines with the same specification as described before forming a Docker Swarm on the Google Cloud Platform. We used the *product* and *product A* service of our sample application running in their own containers as target of all executed release strategies. To collect performance metrics (e.g., CPU utilization, memory consumption), containers hosting cAdvisor and Prometheus were deployed. Moreover, a MongoDB container complemented the deployment setup. While the engine and the proxy had their own VMs, cAdvisor and Prometheus shared the third VM, and the remaining containers shared the fourth VM.

Test Setup. For this experiment the application itself was irrelevant as as long as we could simulate typical engine-to-proxy communication and show the middleware's scalability. Hence, there was no simulated load targeting the case study services during this experiment.

To execute multiple release strategies, we used a slightly modified version of the release strategy presented in Section 4.5.1. The strategy consisted again of 4 phases (canary, dark launch, A/B test, phased rollout) with a duration of 280 seconds in total. The checks and routing instrumentation for `product B` were not relevant for this experiment and were consequently removed. The duration of the final phase was decreased by 100 seconds.

In order to evaluate the scalability of Bifrost with regards to parallel strategies, we increased the number of executed release strategies in a stepwise manner from 1 over 5 to 10, and then for each additional step by 10 until 200 strategies. Our goal was to observe the load on the Docker container running the Bifrost engine, which is responsible for enacting the defined release strategies. A single test run was repeated 5 times, including the collection of CPU and memory utilization data, and the raw duration of each strategy execution, i.e., end time – start time.

Test Results. Figure 4.7 shows the engine's CPU utilization when running multiple strategies in parallel. CPU utilization is the driving factor as both the engine's and the proxy's memory consumption was on a stable, but increasing level. Even though executed on a cheap cloud instance with a single core CPU, the engine is able to handle more than 100 strategies executed in parallel. When considering that even industry leaders in continuous deployment, such as Facebook [Savor et al., 2016; Tang et al., 2015], deploy between 100 and 1000 times a day, this is a good indication that our middleware is able to handle realistic concurrent deployment numbers even on low-end public cloud resources.

This is also supported by looking at how long it takes Bifrost to enact each of those strategies. This is visualized in Figure 4.8. Up to 80 parallel strategies, there is a small, linear increase in delay for each additional strategy. From this point onwards, the engine slowly starts to become overloaded, hence the standard deviation of delays increases and the delay rises with each additional strategy substantially.

It should be noted that our experiment represents a worst case for the Bifrost engine, as all strategies in the experiment were executed at the same time and with identical configuration, thus the periodic reexecution of the checks happened

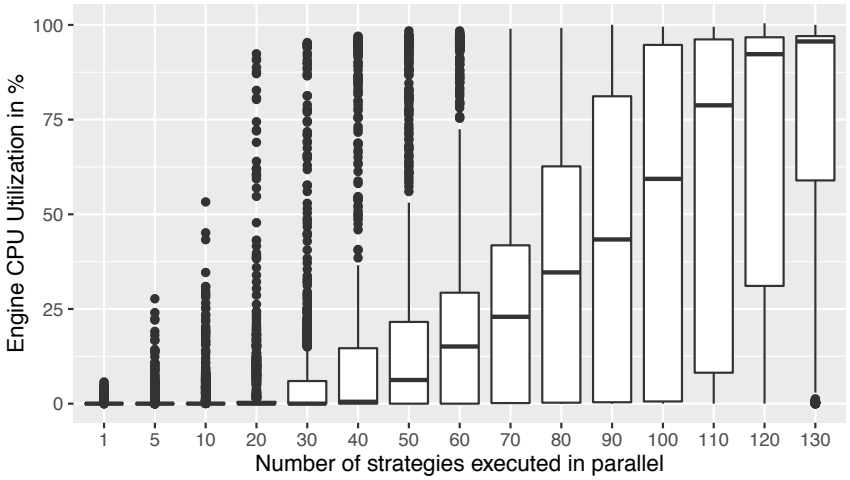

Figure 4.7: Boxplots of CPU utilization on the Docker container running the Bifrost engine. Even with more than 100 strategies being executed in parallel, the instance is rarely fully utilized.

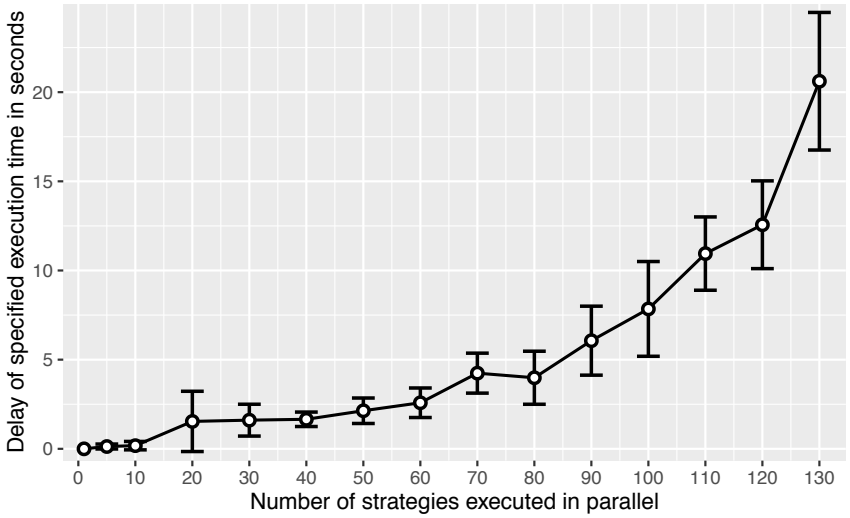

Figure 4.8: Delay in enacting a release strategy when running multiple strategies in parallel. Error bars represent ± one standard deviation.

at the same time as well. However, because of the single core environment, execution at the same time is not possible and thus, a slight delay is introduced for each strategy. However, even in this setting, a delay of 8 seconds in the mean

for enacting 100 releases at the same time is usually negligible in practice, as realistic live testing phases usually span hours or days.

Executing Release Strategies With Many Checks

In this experiment we study the upper bound of parallel checks the Bifrost engine can handle.

Case Study Application Deployment. We launched a cluster of 3 virtual machines forming a Docker Swarm on the Google Cloud Platform, with the same specification as before. Similar to the previous experiment, we focused on the engine's behavior. Hence, no load for the case study application was produced. Besides the engine, we used containers for the *product* and *product A* services, a container hosting a single Bifrost proxy instance, and a container for MongoDB. Moreover, to collect performance metrics (e.g., CPU utilization, memory consumption), containers hosting cAdvisor and Prometheus were deployed. The engine and the proxy instance were deployed on separate VMs, while the remaining 5 containers shared the third VM.

Test Setup. In this experiment, we stressed the engine with a single release strategy, but using an increasing number of parallel checks. Our goal was to identify an upper bound at which the engine is unable to handle the accumulating load. The strategy we used was trivial, consisting only of two identical phases, each running 60 seconds. Each phase contained $8 * n$ checks, where n denotes the current step (stepsize = 10). Out of those 8 checks, 3 target the availability of the product service, and the remaining 5 checks query data from Prometheus. For simplicity, in each step during the experiment, we duplicated the same 8 checks. The engine itself does not cache requests or queries, thus there is no difference whether we would have, for each (re-)execution, queried for different metrics. We repeated each step in our experiment 5 times, and collected CPU and memory utilization data, as well as the raw duration of the strategy's enactment.

Results. As can be seen in Figure 4.9, we were not able to identify an upper limit of checks executed in parallel with our experimental setup. The engine's CPU utilization is slowly increasing for each step. However, even for 1600 checks executed in parallel, we did not reach full utilization. Given the slight increase

for each step and the fact that we executed those checks on a single core machine, this indicates that in a more realistic context with more powerful resources, the engine could even handle higher amounts of checks and thus should be able to cover all realistic monitoring requirements.

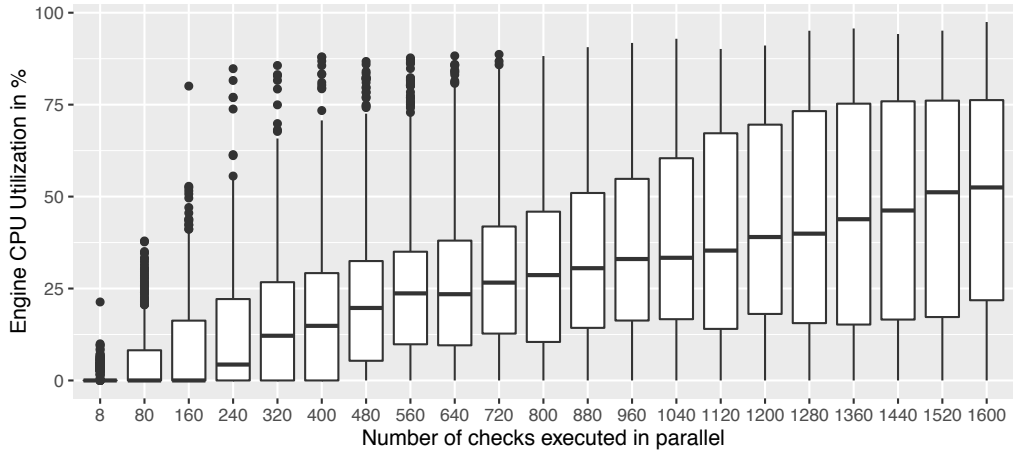

Figure 4.9: Boxplots of CPU utilization when executing an increasing number of checks in parallel for a single strategy.

As in the previous experiment, executing checks at the same time on a single core machine introduces a delay in the enactment of the strategy. This delay increases with the amount of checks executed in parallel and is depicted in Figure 4.10. When executing 1600 checks in parallel, this delay is roughly 50 seconds, which is, given the specified execution time of 120 seconds, quite high. Thus, the delay needs to be taken into account when defining a live testing strategy that uses a very high number of parallel checks. However, arguably, for most practical scenarios a much lower number of checks will be sufficient. In addition, and as before, deploying the engine to a larger cloud instance, specifically one with more virtual CPUs, is likely to mitigate this problem.

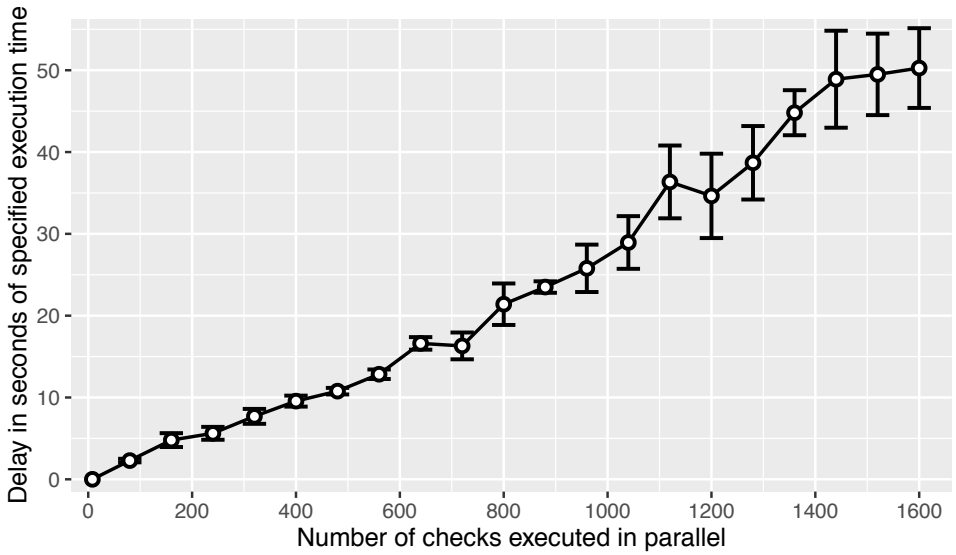

Figure 4.10: Delay in enacting a single release strategy with an increasing number of checks. Error bars represent ± one standard deviation.

4.5.3 Evaluation Summary and Limitations

Our experiments have shown a promising runtime behavior for Bifrost. We have shown that the overhead introduced by using Bifrost for live testing is only around 8 ms for most live testing strategies, even on low-end cloud instances. However, users need to keep in mind that specifically dark launches can substantially increase this overhead due to traffic duplication. We have also shown that our concept and prototype implementation is able to scale to very high numbers of parallel releases as well as parallel checks, indicating that our approach is suitable even for large companies with many parallel rollouts.

The main limitation of our study is that we have only conducted experiments on a single case study application. Hence, we cannot eliminate the possibility that our approach will have higher overhead or scale worse for other applications. Further, while realistic, our case study application was designed specifically for this experiment, and is not a real production application. Secondly, we have conducted our experiments in a virtualized environment (the Google Cloud Platform). It is possible that the performance variations inherent in public

clouds [Leitner and Cito, 2016] have influenced the results of our study. To mitigate this risk, we have repeated each experiment 5 times, and report the observed deviations.

4.6 Related Work

Bifrost as a middleware for automated enactment of live testing strategies in microservice-based systems is strongly related to a number of ongoing trends and developments in modern software development. In an earlier paper, we have already argued for the importance of microservices in modern systems engineering [Schermann et al., 2015]. Cito et al. [2015a] discuss that DevOps [Bass et al., 2015] and data-driven runtime decision making is a core factor in the development of state-of-the-art cloud applications. This has also been confirmed by Begel and Zimmermann [2014], as well as by Kim et al. [2016], who argue that data science is increasingly becoming a central element of the software development and release engineering process. Generally, this newfound interest in data and analytics is related to the current hype surrounding Big Data [Provost and Fawcett, 2013], as well as to the idea of continuous delivery and deployment [Humble and Farley, 2010]. Whereas continuous delivery primarily deals with shortened release cycles, as discussed for instance by Feitelson et al. [2013] for Facebook, continuous deployment goes one step further and largely automates the deployment process [Rodríguez et al., 2016]. Rahman et al. [2015] have studied practices for continuous delivery, and also identified the practices we use (gradual rollouts, dark launches, canary releases, and A/B testing) as central. In our previous work, we have also identified continuous deployment as a prerequisite for live testing [Schermann et al., 2016a]. Conversely, being able to make use of live testing is an important payoff that motivates companies to widely automate their deployment process.

As discussed in Section 4.3.1, a core property of Bifrost is that release decisions are driven by runtime data. Hence, our proposed middleware can build on previous research on application performance management, such as Kieker [van Hoorn et al., 2012] or our own previous work on the monitoring and

management of Web application performance [Cito et al., 2015b, 2014]. Bakshy and Frachtenberg [2015] have recently presented work on statistical methods to identify performance regressions in scale-out cloud systems, based on their experience at Facebook. Another contribution from the Facebook domain [Tang et al., 2015] describes how the company implements gradual rollouts and canary releases. Our basic model of live testing, as discussed in Section 4.3.2, is largely aligned with this description of real-life canary releasing. However, alternative approaches for canary testing, such as CanaryAdvisor [Tarvo et al., 2015], are also available. Another live testing approach for which substantial previous research is existing is A/B testing. Most importantly, Kohavi et al. have proposed a basic model as well as concrete guidelines [Kohavi et al., 2013, 2007] on how to conduct statistically rigorous A/B tests for cloud applications. Tamburrelli and Margara [2014] have rephrased A/B testing as a search-based software engineering problem, which they solve using a combination of aspect-oriented programming and genetic algorithms. Bifrost gives developers a structured way to conduct dark launches, canary releases, or A/B tests, and is fully compatible to the practices described in those earlier works.

In addition, our work is also related to some well-known open source toolkits related to CD and live testing. For instance, the Ruby-based Scientist! framework[9] is a simple library that allows a developer to encode A/B tests directly in code. The disadvantage of this model is that this way live testing code is tangled with the production code base. Further, adapting the configuration (e.g., going from one A/B test to another) requires changes in the application code. A non-intrusive tool that, similarly to Bifrost, builds on top of a microservice architecture to implement A/B testing and canary testing, is Vamp[10]. Unlike our work, Vamp does not support shadow launches or multi-phase rollouts. Another related tool is ION-Roller[11], which focuses on deployment using Docker images. It allows multi-phase rollouts, but only for simple Blue/Green deployment setups. Canary launches require manual monitoring, as it features rollback capabilities upon manual intervention. ION-Roller is a service consisting of an API, web

[9] `https://github.com/github/scientist`
[10] `http://vamp.io/`
[11] `https://github.com/gilt/ionroller`

app and CLI tool that orchestrates Amazon's Elastic Beanstalk to provide safe immutable deployment, health checks, traffic redirection and more. The main advantage of the Bifrost middleware over these existing systems is that it provides developers with a structured way and domain-specific language to arrange and automatically enact multi-phase live testing strategies, a principle that is as of yet largely unexplored.

4.7 Conclusions

In this work, we proposed a formal model for defining live testing strategies covering four previously-identified methods of live testing (canary releases, dark launches, A/B tests, and gradual rollouts). On top of that, we provided a prototype implementation automatically enacting and executing multi-phase release strategies defined in a YAML-based domain-specific language. We evaluated our prototype in three experiments covering (1) the performance overhead introduced to systems when the Bifrost middleware is deployed, and identifying Bifrost's scaling capabilities when confronted with (2) a large number of multi-phase release strategies executed in parallel and (3) release strategies with a large set of continuously evaluated metrics and health checks. Even though our experiments were conducted on cheap public cloud instances, we have shown that the Bifrost middleware adds on average only 8 ms performance overhead when executing a multi-phase release strategy in comparison to a baseline application without Bifrost deployed. The Bifrost's engine is able to handle more than 100 release strategies at the same time on a single core machine and can cope with more than 1000 checks executed in parallel. Hence, we conclude that our approach can be used even in the scale of current-day industry leaders in continuous deployment. Our approach has a number of distinct advantages. Most importantly, formalizing release strategies in a DSL fosters transparency, and allows strategies to be shared, reused, and versioned. Further, additional verification and validation tools can be built on top of our work. While out of scope in this paper, this will be part of our future work.

Additionally, our future work needs to address a number of limitations of the current model and implementation. Most importantly, we are currently not modeling dependencies between services and versions. Similarly, we currently assume that all changes are forward and backward compatible, especially in terms of data schemas. Previous work [Schermann et al., 2016a] has shown that this is not necessarily the case. Finally, we currently assume that provisioning and load balancing service instances is handled outside of Bifrost. Future versions of the tool will be able to instantiate versions themselves, by interfacing with Infrastructure-as-Code tools such as Vagrant or Chef.

4.8 Online Appendix

We provide additional material to this paper, including links to the source code of Bifrost, the case study application used in the evaluation, and a replication package for our study in an online appendix:
`http://www.ifi.uzh.ch/seal/people/schermann/projects/bifrost.html`

4.9 Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610802 (CloudWave), and from the Swiss National Science Foundation (SNF) under project Whiteboard (no. 149450).

5

Topology-aware Continuous Experimentation in Microservice-based Applications

Gerald Schermann, Fábio Oliveira, Erik Wittern, Philipp Leitner
Under submission to an International Systems Conference
Contribution: prototype implementation, experiment design, data collection, data analysis, and paper writing

Abstract

Continuous experiments, including practices such as canary releases or A/B testing, test new functionality on a small fraction of the user base in production environments. Monitoring data collected from previous and new versions of a service is essential for making data-driven decisions on whether to continue or abort experiments. Existing approaches for decision-making rely on service-level metrics in isolation, ignoring that new functionality might introduce changes affecting other services or the overall application's health state. Keeping track of these changes in applications comprising dozens or hundreds of services is challenging. We propose a holistic approach implemented as a research prototype to identify, visualize, and rank topological changes from distributed tracing data. We devise three ranking heuristics assessing how the changes impact the

experiment's outcome and the application's health state. An extensive evaluation on two case study scenarios shows that a hybrid heuristic based on structural analysis and a simple root-cause examination outperforms other heuristics in terms of ranking quality, and a performance evaluation demonstrates that our research prototype is able to analyze service networks with thousands of endpoints within seconds.

5.1 Introduction

The ever-increasing need for rapidly delivering code changes to fix problems, satisfy new requirements, and ultimately survive in a highly-competitive, software-driven market has been fueling the adoption of *DevOps* practices [Bass et al., 2015] by many companies. DevOps promotes the *continuous deployment* [Savor et al., 2016] of code to production, breaking the traditional barrier between development and operations teams and establishing a set of software development methodologies heavily based on tools to automate software builds, integration tests, configuration, and deployment. To further increase development agility, companies are frequently following a *microservice-based* [Newman, 2015] software architecture style. Microservice-based applications comprise a multitude of distributed services, each of which is responsible for a well-defined and typically small functionality. They expose language-agnostic APIs and communicate with each another over the network via HTTP.

The agility facilitated by DevOps practices and micro-service-based architectures enables companies to perform *continuous experiments* [Schermann et al., 2018b], which test the functionality and performance of new versions of application components under production load. A common embodiment of continuous experimentation is to perform *canary releases* [Humble and Farley, 2010]. In this practice, which resembles testing in production, one compares the test version (the "canary") of a microservice against the current version (the baseline) with respect to performance and correctness. Initially, the canary is exposed to requests of a small portion of users. If its performance and correctness remain acceptable, it is gradually exposed to more users until it replaces the baseline. However, if

it fails to perform as expected at any time, all traffic is shifted to the baseline and the canary is terminated. Crucially, determining the health of a canary requires (1) collecting and storing the metrics of interest, and (2) comparatively analyzing the baseline and canary metrics. The comparative analysis is key to decide whether the canary can be safely exposed to more users or should be terminated.

Previous work on assessing the outcome of continuous experiments, and canary releases specifically [Davidovic and Beyer, 2018; Tarvo et al., 2015], considers the microservice under test in isolation, focusing on service-level metrics alone. These approaches ignore the fundamental principle that microservices communicate with each other and that these interactions affect the overall application behavior and can skew, for example, the health assessment of canaries. For instance, when a canary makes a different call or changes call parameters, it can trigger a latent bug in the called microservice. This situation could expose the canary to a delay affecting its own performance metrics, even though the root cause is in the remote service. In this case, changing the canary code under test is likely not the desired corrective action.

Therefore, we contend that continuous experimentation in microservice-based applications must consider the topology underlying all inter-service calls so as to allow developers to evaluate new versions holistically as opposed to in isolation, thereby increasing the confidence in the assessments. It is even more critical to take the overall topology into account when multiple microservices are under experimentation, e.g., running multiple canaries simultaneously.

Given the scale of modern microservice-based applications, often running in the cloud, compounded by a myriad of possible inter-service dependency patterns, distilling topological differences (see Figure 5.2 for an example) and analyzing their impact to effectively guide developers in assessing the outcomes of continuous experiments and the application's health state is a challenging proposition. Out of dozens or even hundreds of identified changes it is crucial to assess those in detail that cause effects on the application's health state. Therefore, we propose an approach to not only identify and visualize changes between baseline and canary versions, but also heuristics to rank these changes based on their potential

impact with the ultimate goal to help assessing continuous experiments. We implemented our approach as a research prototype that supports analyses in the context of multiple experiments running in parallel.

Figure 5.1 depicts the main steps involved in our approach. We infer *interaction graphs* for both the baseline and canary versions from distributed traces collected from microservice-based applications. We compare these interaction graphs to identify topological changes, and rank these changes. A visualization allows developers to review specific changes and associated quality metrics (e.g., response times).

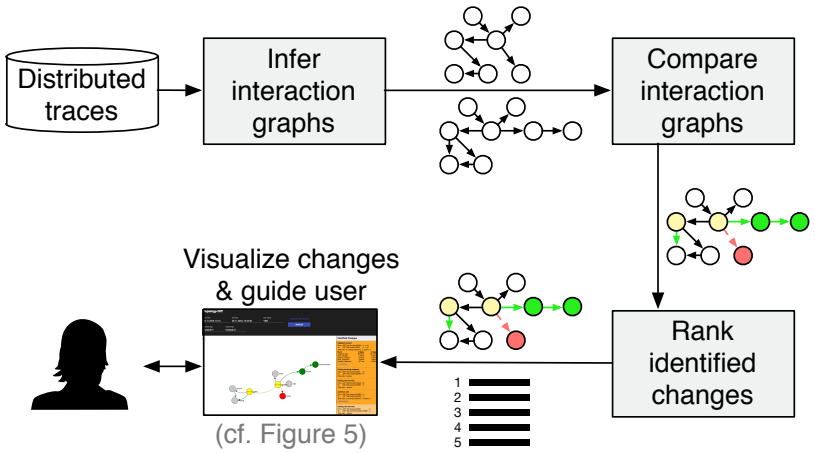

Figure 5.1: Overview of our approach.

In summary, this paper makes the following contributions: (1) a characterization of topological changes that occur in microservice-based applications; (2) a general approach for ranking observed topological changes based on their potential impact on both the experiment assessment and overall application health; (3) three concrete ranking heuristics; (4) a proof-of-concept implementation of the entire approach; (5) an extensive evaluation of the quality of the rankings produced by the heuristics; and (6) a performance evaluation assessing the execution behavior of our approach.

Our evaluation shows that (1) a joint heuristic combining the principles of both structural analysis and performance analysis performs best across multiple

scenarios, and (2) our approach scales well showing promising execution behavior even for applications with thousands of service endpoints.

5.2 Background

Continuous Experimentation. Continuous experimentation is the practice of testing new versions of application components (e.g., microservices), typically with a small portion of the user population and under production load. At the heart of the DevOps *fail-fast* philosophy, its primary goal is to evaluate new code, released frequently and incrementally. Different types of continuous experimentation serve different purposes. In the case of *canary releases* [Humble and Farley, 2010], the new code is compared with a previous version for performance and correctness. Differently, in *A/B testing* [Kohavi et al., 2013], two features are compared with respect to business metrics, e.g., to verify if the new feature increases the revenue generated by a particular product. Finally, in *dark launches* [Kim et al., 2016], production traffic is replicated to a test version without exposing it to users.

Uncertainty in Experimentation. Continuous experimentation helps investigate uncertainties inherent to changing an application. Will the application function as expected after the change? And will it do so in a performant, reliable manner? Arguably, the more disrupting the changes the higher the uncertainty. Changing only the internals of a microservice's implementation, without affecting how it is consumed and the external calls it makes to other microservices, introduces less uncertainty than deploying a completely new microservice and having others call it. This work aims to make different uncertainty degrees explicit in the process of assessing the results of continuous experiments.

Distributed Tracing. To characterize changes that lead to uncertainties, this work relies on distributed tracing, a technique that can be used to collect information about calls between microservices. A *trace* is a set of data about the sequence of all inter-service calls resulting from a top-level action performed by an end user. Each call is associated with four timestamped events corresponding

to sending the request, receiving the request, sending the response, and receiving the response. Traces also contain status information indicating success or failure of each call. Hence, from the data in a trace we can estimate (1) how much time each request and response spent traveling over the network, and (2) how long it takes each microservice to process them. One way to identify the sequence of calls belonging to each trace is proposed by the Open Tracing project Open Tracing [2019] and used by tracing systems such as Jaeger [2019] and Zipkin [2019]. The idea is to create a new trace and assign a unique identifier to it when an external call is made to an edge service. Afterwards, the identifier is propagated on every subsequent call. *Service meshes* such as Istio [2019] and Linkerd [2019] use network proxies colocated with each service to intercept all incoming and outgoing traffic, and store traces in a central database.

5.3 Related Work

Previous research has empirically assessed continuous experimentation practices and challenges [Schermann et al., 2018a,b]. These works analyze reports on continuous experimentation practices by selected companies [Fabijan et al., 2017; Kevic et al., 2017; Tang et al., 2015], and also present data collected more broadly using interviews and surveys. They find that software architectures based on components that can be deployed and operated independently (e.g., microservices) are essential for continuous experimentation, but also attest that root-cause analysis of observed problems is challenging. Our work attempts to address these challenges by considering the interactions in which updated services participate.

Multiple methods and systems have been proposed for continuous experimentation. *Kraken* is a system proposed by Facebook [Veeraraghavan et al., 2016] for traffic routing between services, servers, or even data centers to identify performance bottlenecks using actual user traffic. *Bifrost* [Schermann et al., 2016b] formalizes continuous experiments consisting of multiple phases. A proposed domain-specific language allows developers to design experiments, which are then automatically executed by a middleware using smart traffic routing. The *MACI*

framework [Froemmgen et al., 2018] for management, scalable execution, and interactive analysis presents an alternative way to express experiments integrating recurring tasks around experiment documentation and management, scaling, and data analysis with the goal of reducing specification efforts.

The work by Sambasivan et al. [2011] is the closest to our approach. It compares distributed traces to diagnose performance changes, distinguishing between *structural* changes and ones in *response-time*. While Sambasivan et al. assume similar workloads for the variants, our approach focuses on the topology and on experimentation settings to assign only a small fraction of users to experimental variants. In our approach the mapping between the variants (i.e., our set of change types) is more fine-grained as we compare traces at the HTTP endpoint, version, and service levels.

Finally, our work relies on distributed traces collected by the Istio service mesh [Istio, 2019] to infer topologies of microservice-based applications. A number of distributed tracing systems have been developed. *Dapper* [Sigelman et al., 2010] relies on annotating messages (also proposed by X-Trace [Fonseca et al., 2007]) to combine a request sequence into a trace. Open-source systems such as Jaeger [2019] and Zipkin [2019] build on the design proposed by Dapper.

5.4 Characterizing Change Types

In the following, we characterize reoccurring change types we identified when comparing service topologies. For this purpose, we derive formal representations of microservice-based applications and service-interaction graphs that frame our basis to define topological change types.

5.4.1 Microservice-based Application

A microservice-based application $\mathcal{A}$ consists of a set of interacting services $\mathcal{A} = \{s_1, s_2, \ldots, s_n\}$. Services are available in different versions, e.g., stable version 1 of the *frontend* service and a new experimental canary version 2 depicted in Figure 5.2. For a service $s_i \in \mathcal{A}$ this is represented as a tuple

$\mathcal{VS}_i = \langle s_{i,1}, s_{i,2}, \ldots, s_{i,n} \rangle$, where $s_{i,1} \ldots s_{i,n}$ are the corresponding versions j of service s_i with $1 \leq j \leq n$. Note that Figure 5.2 not only represents our running example, but also depicts a topological difference which we will cover in detail in later sections when we revisit this example.

In the context of continuous experiments a microservice-based application is available in multiple *variants* $\mathcal{VA} = \langle va_1, \ldots, va_n \rangle$ at the same time. An application variant comprises a combination of services $\langle s_i, \ldots, s_k \rangle$ with $i \leq j \leq k$ and $s_j \in \mathcal{A}$. For each of those services $s_j \in \mathcal{A}$ a concrete version u with $s_{j,u} \in \mathcal{VS}_j$ is selected. In Figure 5.2, the *baseline* variant of the application includes version 1 of *frontend*, while the *canary* variant includes the new version 2 of *frontend*. Every service version $s_{i,j}$ (i.e., version j of service s_i) offers service endpoints $\mathcal{EP}_{i,j} = \langle s_{i,j,1}, s_{i,j,2}, \ldots, s_{i,j,k} \rangle$, e.g., `HTTP` endpoints such as `'POST /orders/{productId}'` and `'GET /orders'`. The endpoints $s_{i,j,1} \ldots s_{i,j,k}$ implement the concrete functionality provided by $s_{i,j}$.

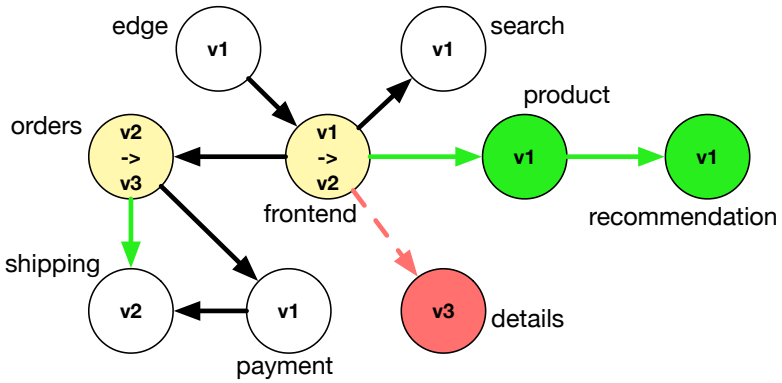

Figure 5.2: Topological difference of a microservice-based sample application. Green depicts added functionality or calls, red depicts removed functionality or calls, and yellow depicts service version updates.

5.4.2 Interaction Graph

In a microservice-based application, version j of a service s_i interacts with other services by calling one or more of their endpoints. In our model, this interaction is represented by a directed graph $G = \langle V, E \rangle$ in which V and E denote sets of vertices and edges respectively. Every endpoint $s_{i,j,k}$ of an application corresponds to a vertex $v \in V$ in the graph, referring to version j of $s_i \in \mathcal{A}$, where $s_{i,j} \in \mathcal{VS}_i$. A directed edge $e = s_{i,j,k} \rightarrow s_{u,v,w}$, where $e \in E$, represents a call from a service endpoint $s_{i,j,k}$ (subsequently named *caller*) to another service endpoint $s_{u,v,w}$ (subsequently named *callee*). A service endpoint $s_{i,j,k}$ itself does not call another service's endpoint. Rather, the call is made by version j of service s_i in the execution context of the code associated with $s_{i,j,k}$. However, for constructing an interaction graph and thoroughly reasoning about change impact, we model endpoints as sources and targets of inter-service calls.

$\mathcal{I}_{i,j,k}$ denotes a set of *inbound* calls for a service endpoint $s_{i,j,k} \in V$ containing all vertices $s_{m,n,o} \in V$ for which an edge $e \in E$ with callee $s_{i,j,k}$ exists, i.e., $e = s_{m,n,o} \rightarrow s_{i,j,k}$. Similarly, $\mathcal{O}_{i,j,k}$ denotes a set of *outbound* calls for a service endpoint $s_{i,j,k} \in V$ containing all vertices $s_{m,n,o} \in V$ where there exists an edge $e \in E$ with $s_{i,j,k}$ as the caller, i.e., $e = s_{i,j,k} \rightarrow s_{m,n,o}$.

To simplify the presentation within this paper, we will visualize examples on a service level rather than on the endpoint level (e.g., Figure 5.2). To stay consistent with the definitions provided later, assume that every service only exposes a single endpoint, resulting in a single vertex in the interaction graph. However, this is only used for presentation purposes. Our approach supports services with multiple endpoints.

5.4.3 Topological Change Types

The presented formal model allows us to construct interaction graphs for every application variant and to compare them. Comparing interaction graphs of two or more variants reveals changes at the topological level. For example, in Figure 5.2, when the canary version 2 of *frontend* is deployed, we observe that a new service (*product*) is required while the *details* service is no longer called.

In the following, we characterize typical change types that surface in the evolution of microservice-based applications. When comparing interaction graphs G_1 and G_2, every such change type appears as a certain pattern involving a subset of the vertices. We distinguish two categories of change types: *fundamental* and *composed*, where a *composed* change type is a combination of multiple *fundamental* change types.

Fundamental Change Types

Fundamental change types involve calling newly added services (or service endpoints), calling endpoints of existing services, or removing calls to service endpoints.

Calling a New Endpoint. This change type represents new functionality manifesting as a call to a new resource, such as a service or a service endpoint that was added. In both interaction graphs G_1 and G_2 there exists a vertex (or node) representing an endpoint m of a service a, but in different service versions: i in case of G_1 (i.e., $s_{a,i,m}$), and j in case of G_2 (i.e., $s_{a,j,m}$). The interaction graph of G_2 contains an edge $e \in E$ with $e = s_{a,j,m} \rightarrow s_{u,v,w}$ calling a service u in version v that does not exist in the interaction graph of G_1. Figure 5.3 (left) depicts this change type in our running example. The endpoint of the *frontend* service of the *canary* variant (version 2) calls a newly added *product* service that does not exist in the *baseline* variant (version 1).

Calling an Existing Endpoint. This change type characterizes reusing functionality, i.e., a new call to an existing service endpoint is made. There are again two nodes in the interaction graphs representing the same service a with endpoint m, but in different service versions: $s_{a,i,m}$ in G_1 and $s_{a,j,m}$ in G_2. The interaction graph of G_2 contains an edge $e \in E$ with $e = s_{a,j,m} \rightarrow s_{u,v,w}$ denoting a call to service u that also exists in the interaction graph of G_1; thus, $s_{u,v,w}$ is represented by a vertex $v \in V$ of G_1. However, there is no direct interaction (no edge) between $s_{a,i,m}$ and $s_{u,v,w}$ in G_1. Figure 5.3 (center) shows this change type in which the *canary* variant of *orders* (version 3) calls *shipping*. The *shipping* service is also part of the *baseline* variant involving version 2 of *orders*, but there is no direct interaction between *orders* and *shipping*.

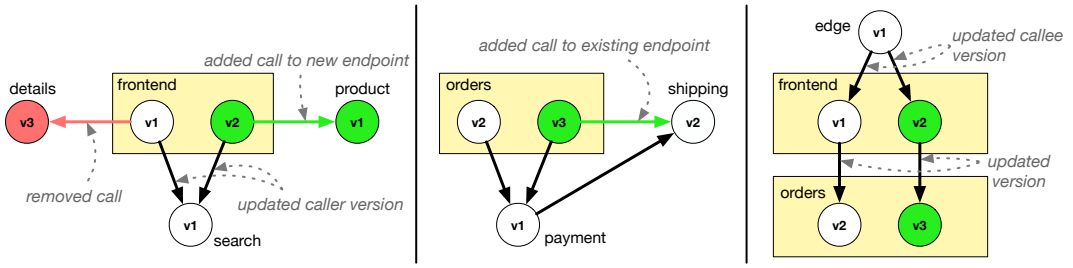

Figure 5.3: Topological change types demonstrated on sample application (excerpt). Left: add call to new service, removed call, and updated caller version. Center: add call to existing endpoint. Right: updated callee version and updated version.

Removing a Service Call. This change type represents the inverse of the previous one. In this case, a previously used resource (an entire service or a service endpoint) is no longer used. Revisiting the previous change type, this time the interaction graph of G_1 contains an edge $e \in E$ with $e = s_{a,i,m} \rightarrow s_{u,v,w}$ representing a call to a service u, but no equivalent edge between $s_{a,j,m}$ and $s_{u,v,w}$ exists in G_2. However, the service u (and its endpoint m) might still be used in G_2 by other services. Figure 5.3 (left) represents this change type between the *canary* variant of *frontend* (version 2) which no longer calls *details*.

Composed Change Types

These change types are constructed from fundamental change types and denote updated caller version, updated callee version, and updated version.

Updated Caller Version. When comparing interaction graphs G_1 and G_2, the version of a calling service a is "updated". This caller-side version update is a combination of *removing a service call* and *calling an existing endpoint* change types. From the perspective of G_2, the endpoint $s_{a,i,m}$ no longer calls a service endpoint $s_{u,v,w}$ (i.e., removed service call), but the same endpoint m of the updated service version ($i \rightarrow j$) is adding a call to $s_{u,v,w}$ (i.e., calling an existing endpoint). This change type requires that exactly the same endpoint m of the same service a but of a different version ($i \neq j$) calls exactly same endpoint $s_{u,v,w}$. Figure 5.3 (left) depicts an example. In the canary variant, the

frontend service is updated to version 2, and both version 1 and version 2 call the *search* service.

Updated Callee Version. This change type represents the case of a version change in the service that is called. This callee-side version update combines *removing a service call* and *calling a new endpoint* change types. From the perspective of G_2, the endpoint $s_{a,i,m}$ no longer calls a service endpoint $s_{u,v,w}$ (i.e., removed service call), but the same endpoint $s_{a,i,m}$ calls a new version of service u's endpoint w (version update: $v \rightarrow x$, i.e., calling a new endpoint), hence there exists an edge $e = s_{a,i,m} \rightarrow s_{u,x,w}$. Figure 5.3 (right) exemplifies this change type when the version of *frontend* that is called by *edge* is updated from version 1 (*baseline*) to version 2 (*canary*).

Updated Version. This change type is a combination of *updated caller version* and *updated callee version* change types. There exists an endpoint m of service a and an endpoint w of service u in both interaction graphs G_1 and G_2. In G_1 there exists an edge $e_1 = s_{a,i,m} \rightarrow s_{u,v,w}$, and in G_2 there is an edge $e_2 = s_{a,j,m} \rightarrow s_{u,x,w}$. Hence, in G_1 the interaction happens between versions i and v of the services a and u, and in G_2 between versions j and x. From the perspective of G_2, both the caller and the callee versions are updated. This pattern applies if and only if both caller and callee services stay the same, including the endpoint, with only the service versions being updated. This type of pattern often surfaces in the case of breaking changes within the endpoint functionality such that an update on the called side causes an update on the calling side. Figure 5.3 (right) shows this pattern on the interaction between *frontend* and *orders*. While for the *baseline* variant the interaction happens between version 1 of *frontend* and version 2 of *orders*, the *canary* variant requires version 2 of *frontend* and version 3 of *orders*.

5.5 Ranking Identified Changes

This section covers (1) the construction of the graph-based topological differences, (2) a generic algorithm that traverses these differences to produce a ranking of

identified changes, and (3) three embodiments of this algorithm in the form of heuristics to assess the impact of the changes identified.

5.5.1 Constructing the Topological Difference

Our approach relies on distributed traces of a microservice-based application to (1) infer interaction graphs for each variant of the experiment and to (2) construct a graph-based topological difference resulting from their comparison.

Inferring Interaction Graphs. Our prototype relies on distributed tracing data collected by the Istio [2019] service mesh using either Zipkin [2019] or Jaeger [2019]. In order to assess the outcome of an experiment, a developer needs to first identify the application variants of interest. In the case of a canary release, for instance, versions of the services for baseline and canary need to be provided. Another required input is the experiment start time. Given the inputs, the first action executed by our continuous experimentation assessment system is to divide baseline and canary traces into clusters, where each cluster contains multiple interaction graphs (as defined in Section 5.4) with the same *root request*. A *root request* is a service endpoint call made to an edge service of the application under experimentation, which in turn triggers other inter-service calls within the application, forming an interaction graph. In each cluster we also compute statistics on metrics for each inter-service call, namely, duration, timeouts, retries, and errors. For instance, in a cluster based on our running example, the edge denoting the call from *orders* to *payment* would batch together multiple such calls over which the aforementioned statistics are computed.

Comparing Interaction Graphs. The next step is to compare corresponding baseline and canary clusters of interaction graphs to identify topological changes based on the types described in Section 5.4.3. Once the changes and their types are identified, for analysis and visualization purposes, the graphs are merged into a single graph forming an "extended" topological difference (e.g., Figure 5.2 for the running example). Nodes and edges in this resulting topological difference are annotated accordingly to keep track of their original interaction graphs. The topological difference contains all the changes identified, their assigned type, and further statistics that were captured during the interaction

graph's construction. Due to the merge, the difference graph contains also those structures (endpoints and their interactions) that are common to the graphs under comparison. Doing so preserves the "big picture" and enables detailed analyses on the entire service network. Furthermore, it makes it easier to spot deviations (e.g., performance problems) that are not directly associated with a change but that may result from the cascading nature of service calls (e.g., a *common* service endpoint is called more frequently in the canary variant leading to a higher response time).

5.5.2 Traversing the Topological Difference

Once the graph-based topological difference is built, we execute a two-phase graph-traversal algorithm, consisting of the *annotation* and the *extraction* phases.

Basic Algorithm. In a first step, all endpoints (i.e., nodes in the graph) without *outbound* calls are visited (and marked as such). Then, the algorithm visits those endpoints calling service endpoints that have been flagged as visited, marking them as visited again. This process is repeated until all nodes in the graph are visited. Single captured traces that contain multiple calls to the same endpoints could result in cycles in the graph. These cycles are "broken up" by visiting nodes based on the reversed temporal order of the endpoint calls, which is extracted from the tracing data.

Annotation Phase. In our approach, every node in the graph-based topological difference has an associated state $\mathcal{T}$, which is used to store any information to reason about, and ultimately rank changes. In the *annotation* phase, these states are set to hold information required for the concrete implementation of the ranking algorithm (i.e., heuristic). During a node's visit, a wide range of information is available, including the involved endpoint, outgoing calls and their change types, statistics (for either one or for both variants) that were computed during the construction of the interaction graphs, and any other queryable monitoring information (e.g., from Prometheus [2019] instances). It depends on the concrete implementation of a heuristic which information is used and how it is combined. This allows revealing different insights on the experiment's outcome

and the application's health state. In the scope of this paper we cover three heuristics (see Sections 5.5.3, 5.5.4, and 5.5.5) in various combinations.

Extraction Phase. In this phase, every node is revisited with the goal to *extract* a score $\mathcal{S}$ for each interaction (i.e., outgoing edge). Due to the nature of our change types, an interaction in the topological difference graph could comprise two edges in the source interaction graphs. The scoring happens on the change type level: edges belonging to the same change are merged. Edges that are common (without any change) in both source interaction graphs are treated as a special change type. The idea of the extraction phase is to rely on the state information gained in the annotation phase and to transform it into scalar values. Formally, this scoring function has the type signature $score : change \rightarrow int$.

Ranking. Once scores for all edges in the difference graph are computed, the scores are sorted in descending order and ranks from 1 to k are assigned, where k is the number of edges in the graph-based topological difference. The edge achieving the highest score is ranked on position 1. Equal scores leading to tied ranks are possible, even though they appear rarely.

In the following we will cover three specific embodiments of our algorithm. Starting with the *Subtree Complexity* heuristic, followed by the *Response Time Analysis* heuristic, we will cover their joint variant, the *Hybrid* heuristic.

5.5.3 Subtree Complexity Heuristic

This heuristic analyzes sub-structures of a topological difference and considers uncertainty in the context of experiments.

Concept. The graph structure is broken down into multiple subtrees (see Figure 5.4 for an example). The fundamental idea of this heuristic is that the more complex the structure of the (sub-)tree is, the more likely it contains changes that affect the outcome of the experiment and the application's health state.

Initially, every node a has an assigned state of $\mathcal{T}_a = 0$. Whenever a node a is visited during the algorithm's annotation phase, its state $\mathcal{T}_a$ is set to $\mathcal{T}_a = \sum_1^n \mathcal{T}_i + p_{a,i}$ being $1 \leq i \leq n$ the (child) nodes of the outgoing calls of a. Thus, the state values $\mathcal{T}_i$ of called nodes i are summed up and weights $p_{a,i}$

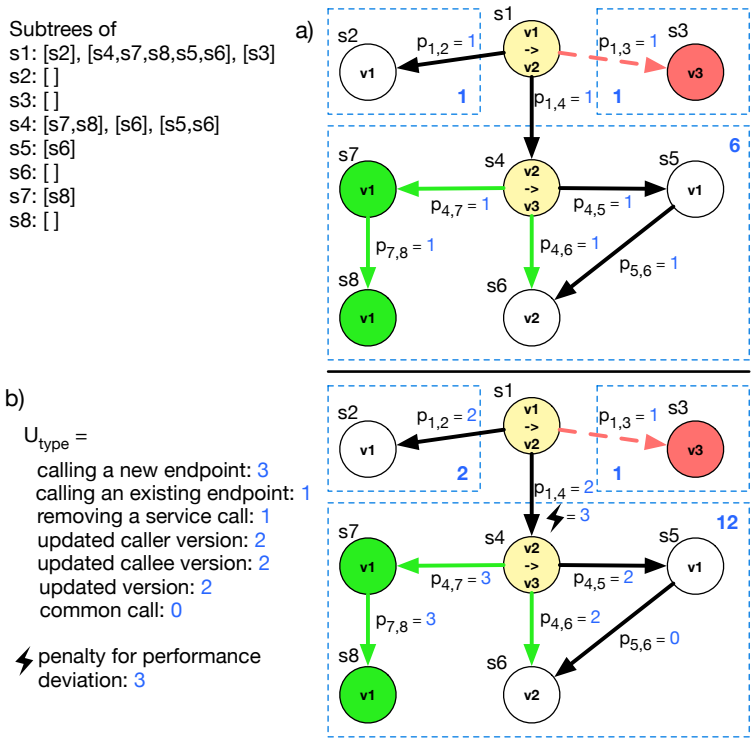

Figure 5.4: Example of (topmost) subtrees in a topological difference. **a)** Basic subtree complexity (ST) in blue (i.e., counting the number of edges in a subtree). Service $s1$ has three subtrees. The state value of $s4$ is 5. Thus, the extracted score for the edge between $s1$ and $s4$ is $5+1=6$. **b)** Extended subtree (ST Ext) in blue, propagation values $p_{a,i}$ based on U_{type} values assigned to change types. Extracted score for the edge between $s1$ and $s4$ is $10+2+3=15$. (3 represents the performance penalty).

representing individual *propagation* factors for these calls are added. During the *extraction* phase, for every interaction of a node a with a node i, the score for this edge e is computed as follows: $\mathcal{S}_e = \mathcal{T}_i + c_{a,i}$. Thus, the score is built from the state value $\mathcal{T}_i$ of the node (i.e., service endpoint) that is being called and an individual *scoring* factor $c_{a,i}$ for the edge.

The distinction between *propagation* and *scoring* factors serve the following purposes. The propagation factor directly influences the state values of the nodes (and thus the individual scores) when walking up the tree. This is useful if severe

issues within a subtree are detected that should be reflected in the ranking of the changes. The scoring factor only influences individual scores, e.g., a single change. It allows expressing fine-grained differences among the changes. Depending on how propagation and scoring factors are chosen, the subtree complexity heuristic allows for multiple variations. Within the scope of this paper, we focus on two variations: *Subtree* and *Subtree Extended*.

Subtree (ST). This standard variant of the heuristic analyzes the structural complexity of the difference graph by counting the number of interactions (i.e., edges) within subtrees. Propagation and scoring factors $p_{a,i}$ and $c_{a,i}$ are set to 1 for all edges independent of their change types. Figure 5.4a depicts an example in blue.

Extended (ST Ext). This variation introduces the concept of *uncertainty*. Calling entirely new services compared to calling a new version of an existing service leads to a different degree of uncertainty when assessing the application's health state. For the former, no information to compare to (i.e., previous calls or historical metrics) exists, while for the latter calls to the new version can be compared with previous calls. Deviations in metrics, such as response times or error rates, can be considered. Similarly, when a new call to an existing endpoint is made, even though a direct comparison on the interaction-level is not possible, there are still metrics available that are associated to the called service allowing an assessment whether this added call introduces unwanted effects. In our approach, we built upon these subtle differences in uncertainty for the identified change types and assign a weight U_{type} to each of them.

For the *extended subtree* heuristic, instead of the number of edges, the uncertainty values U_{type} associated to the individual edges' change types are summed up within a subtree. Hence, individual propagation factors $p_{a,i} = U_{type}$ are set to the uncertainty value of the edge's change type. Figure 5.4b depicts an example. The rationale for this is to emphasize the uncertainty of subtrees involving many changes. Scoring factors are defined as $c_{a,i} = U_{type} + P$. Similar to the propagation factors we use the uncertainty values U_{type} and we introduce penalties P that are added to those interactions for which deviations are measured, e.g., significant changes in response times. This mechanism allows us to account

for performance issues without running in depth root-cause analyses. Penalization applies to all interactions for which direct comparisons between the variants on the edge-level are possible, i.e., composed change types and common calls.

5.5.4 Response Time Analysis Heuristic

This heuristic pays attention to performance deviations. It tries to identify services and changes that have caused performance issues by incorporating the notion of uncertainty.

Concept. The intuition here is that in case of performance deviations (e.g., response time) spotted at a node, the node's surrounding changes that add additional calls (e.g., *calling a new endpoint*, or *calling an existing endpoint*) are potential sources of these deviations. This heuristic focuses on the overall response time (i.e., how long did the called endpoint take to respond) extracted from tracing data. However, the concept can be extended to incorporate other metrics that have similar cascading effects. Further, note that these performance comparisons are only possible for specific change types, namely composed change types and common calls.

The state $\mathcal{T}_a$ of a node a is extended to keep track of deviations and their potential sources while traversing the graph. It involves *flag*, a counter that keeps track how often a node is considered as the source of a deviation, a map *deviations* that stores which outgoing call (i.e., key) causes how much deviation (i.e., value, in milliseconds), and a list *source* keeping track which child caused the deviation. Algorithm 1 illustrates the analysis executed for every outgoing call in the *annotation* phase when visiting a node a.

In case of a deviation, the called child is added as a source. If there are no stored deviations for the child node, then the deviation is added to the node's state, and the child's state *flag* counter is set to 1. If there are deviations, the recursive function *flagSources* walks through all the stored sources that might caused the deviation on the child's side and increases their *flag* counters. In the next step, the sum of all stored deviations (i.e., *total*) is calculated and the deviation is added to the node's state. If the call's deviation is higher than the total sum of deviations on the child's side, then it is likely that a change

Algorithm 1: Response Time Analysis

```
Input: node, child, call
if call.hasDeviation() :
    node.state.addSource(child)
    if len(child.state.deviations) == 0 :
        node.state.addDeviation(call=call,deviation=call.deviation)
        child.state.flag := 1
    else:
        flagSources(child)
        total := sum(child.state.deviations)
        node.state.addDeviation(call=call, deviation=max(call.deviation, total))
        if call.deviation > total :
            inc(child.state.flag)
            for c in child.calls :
                if c.type in [call_new_endpoint, call_existing_endpoint] :
                    inc(c.target.state.flag)
                    child.state.addSource(c.target)
```

introduced this new deviation. Therefore, the child's *flag* counter is increased and the child's surrounding changes are analyzed. This involves all of the child's outgoing edges with *calling a new endpoint* and *calling an existing endpoint* change types. The target nodes of these edges are added as potential sources and their flag counters are increased.

By using different *scoring* factors in the heuristic's *extraction* phase we distinguish two variations: RTA and RTA Ext. The *annotation* phase (i.e., flagging) described in Algorithm 1 is the same for both variations.

Response Time Analysis (RTA). This represents the heuristic's standard variant. In the *extraction* phase, for every outgoing call of a node a to a child node i, the score for an edge e is defined as $\mathcal{S}_e = \mathcal{T}_{i.flag}$. The resulting score corresponds to the final value of the child node's *flag*. Consequently, those service endpoints with the highest flag counts are ranked first.

Extended (RTA Ext). For this variation we revisit the concept of uncertainty and reuse weights U_{type} as scoring factors. Again, the rationale is that those interactions with high uncertainty for a change should have higher scores. To have a mechanism to balance between *flag* and *uncertainty* values, we introduce a *penalty* constant C. The scoring function for an edge e is defined as $\mathcal{S}_e = \mathcal{T}_{i.flag} * C + U_{type}$.

5.5.5 Hybrid Heuristic

More complex (sub-)structures are more likely to contain changes that could cause problems. This is the strength of the subtree complexity heuristic. However, in case of performance deviations, the response time analysis heuristic provides more detailed analyses to identify the origin of problems. The goal of the hybrid heuristic is to combine the strengths of both, structural and performance analyses.

The underlying mechanics of both heuristics remain untouched for the hybrid heuristic. During the algorithm's annotation phase, both the structural and the performance analyses are conducted. The extraction phase shapes how the individual results of both heuristics are transformed into a single result. We distinguish two variants:

Hybrid (HYB). The standard variant of the hybrid heuristic uses the *extended subtree* heuristic (ST Ext) to determine state values $\mathcal{T}_i$ and the *standard RTA* variant to determine *flag* values. Consequently, the scoring function for an edge e is defined as $\mathcal{S}_e = \mathcal{T}_i + U_{type} + \mathcal{T}_{i.flag}$.

Extended (HYB Ext). The extended variant of the hybrid heuristic is based on the *extended subtree* heuristic (ST Ext) for determining state values $\mathcal{T}_i$ and the *extended RTA* variant to determine *flag* values. Hence, the scoring function for an edge e is defined as $\mathcal{S}_e = \mathcal{T}_i + U_{type} + \mathcal{T}_{i.flag} * C$, being C the penalty constant established in RTA Ext.

5.6 Implementation

To demonstrate our (formal) approach we developed a research prototype with the goal to assist developers on experiment health assessment and decision-making. The server-side backend implements (1) the trace analysis (i.e., gathering traces from distributed tracing systems, clustering these traces, and inferring interaction graphs) and the change type identification in Python, and (2) the heuristics that rank identified changes as a TypeScript-based Node.js application. Our client-side frontend, which is implemented in TypeScript in combination with React, interactively visualizes the difference graphs alongside a ranking of these

changes which is produced by the selected heuristic. Color coding based on the resulting scores highlight changes that are considered important for further manual inspection by the developer.

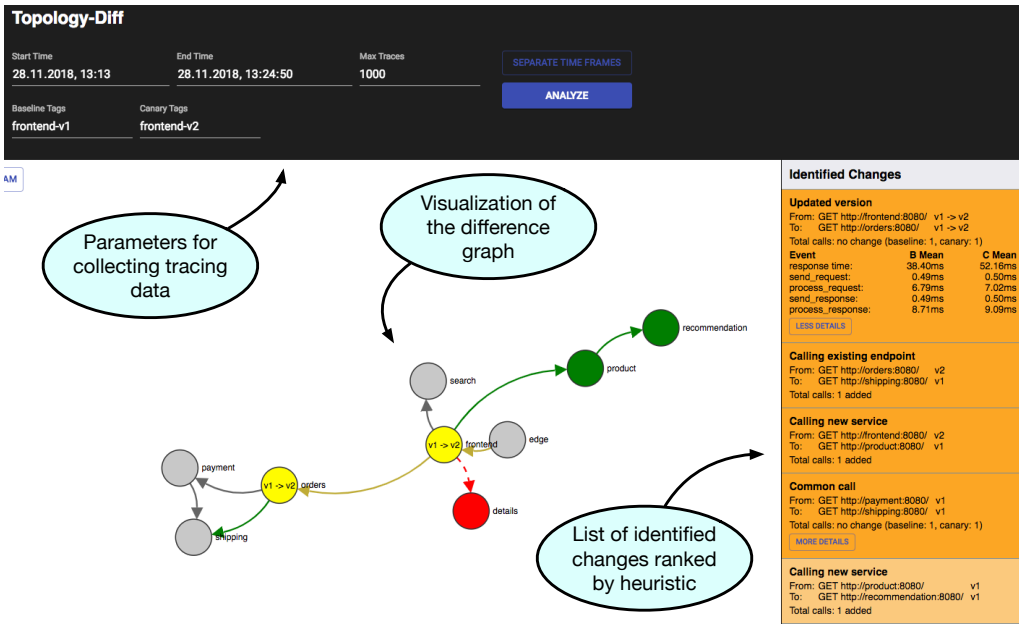

Figure 5.5: Screenshot of our research prototype visualizing the difference graph of the running example.

Figure 5.5 shows a screenshot of our frontend. The derived difference graph is displayed in the center, the list of ranked changes on the right. Additional information (e.g., response times, error rates) attached to the ranking entry can be explored on demand to support decision-making. More and higher resolution screenshots of our UI, and the heuristics' source code can be found in our paper's online appendix [Schermann et al., 2019].

5.7 Ranking Quality Evaluation

We assessed our approach in two dimensions: a ranking quality evaluation (current Section) and a performance evaluation (Section 5.8). The former was conducted on two concrete scenarios: (1) revisiting the running example, and (2)

dealing with multiple breaking changes. The paper's online appendix Schermann et al. [2019] provides a comprehensive replication package. Before we dive into details of the ranking quality evaluation, we briefly describe our evaluation's setup.

5.7.1 Setup

The setup involves a description of the method we used to assess the quality of the produced rankings, how we calibrated the parameters the heuristics are operating on, and how we generated the distributed tracing data.

Method. Normalized discounted cumulative gain (nDCG) [Järvelin and Kekäläinen, 2002] is a measure of ranking quality, widely used in information retrieval, but also an established metric in other fields (e.g., test case prioritization [Mostafa et al., 2017]). Based on a graded *relevance* scale of documents in the result list of search-engine queries, DCG (or its normalized variant nDCG) assesses the usefulness (i.e., the gain) of a document based on its position in the result list. The gain of each document is summed up from top to bottom in the ranking, having the gain of each result discounted the lower the rank, which has the consequence that highly relevant documents ranked at lower positions are penalized. The DCG accumulated at a particular rank position p is defined as follows:

$$DCG_p = \sum_{i=1}^{p} \frac{rel_i}{log_2(i+1)}$$

rel_i is the relevance of the document at position i. Instead of documents we rank identified changes. In order to use DCG, the authors assessed the relevance of every single change of our two scenarios. In total, including sub-scenarios, 6 relevance assessments were conducted rating changes on a scale from 0 (not relevant) to 4 (highly relevant). We use a normalized DCG (nDCG) returning relative values on the interval 0.0 to 1.0, this allows for result comparison across scenarios. 1.0 is the maximum value representing a ranking with the most relevant changes on the top positions. As tied ranks are possible (e.g., changes with the same score and rank as resulting from a heuristic), we applied the nDCG

adaption proposed by McSherry and Najork [2008] considering average gains at tied positions.

Calibration. To calibrate the heuristics we followed an iterative exploratory parameter optimization procedure across all scenarios. For nDCG we considered the top $3, 5, 7$, and 10 positions of the ranking to be compared. For the penalties P and C used in the heuristics' scoring functions we iterated through values $1, 3, 5, 7$, and 10. We tested four different mappings of *uncertainty* values to change types U_{type}. Based on more than 9000 calibration results, we determined that $P = C = 3$ and an uncertainty mapping U_{type} (i.e., *change type* $\rightarrow$ *uncertainty*) of {'calling new endpoint' : 3, 'calling existing endpoint' : 1, 'removing call' : 1, 'updated caller version' : 2, 'updated callee version' : 2, 'updated version' : 2, 'common call' : 0} yielded the most promising results. We determined the nDCG for the top 5 positions to allow comparison across scenarios of different sizes.

Tracing Data. We implemented the two evaluation scenarios as microservice-based applications running on top of a Kubernetes cluster in the IBM Cloud. The Istio service mesh was in place to handle experiment traffic routing between the application's variants along with a ZipKin installation keeping track of service interactions. Every service in the evaluation scenarios exposed a single `HTTP` endpoint. For every (sub-) scenario 1000 requests were generated, 70% routed to baseline and 30% to canary variants. In general, 30% for a canary is uncommon. We only used it for evaluation purposes to collect more data points (i.e., canary traces) faster with less overall requests.

5.7.2 Scenario 1: Revisiting the Sample Application

As a first scenario we use the example application shown in Figure 5.2. Contrary to the next scenario, we do not cover a specific evaluation aspect here. However, this scenario involves all of the change types we identified, hence making it a useful baseline to assess the proposed heuristics.

Scenario. This scenario involves two sub-scenarios: *basic* and *delayed*. *Basic* executes the baseline variant of the application without modification, the canary variant involves added functionality and updated service versions. The *delayed* sub-scenario introduces a delay of $100ms$ at the *payment* service for the *canary*

variant. This reflects an abnormally behaving *orders* service in the *canary* that multiplies the traffic towards the *payment* service causing it to overload, resulting in higher response times.

Relevance. For the basic scenario, the added calls to *product* and the updated versions of *frontend* and *orders* were classified as highly relevant (i.e., a relevance score of 4). For the delayed scenario, in addition, the call between *payment* and *orders* is classified as highly relevant. Our relevance ratings for all scenarios are listed in our online appendix Schermann et al. [2019].

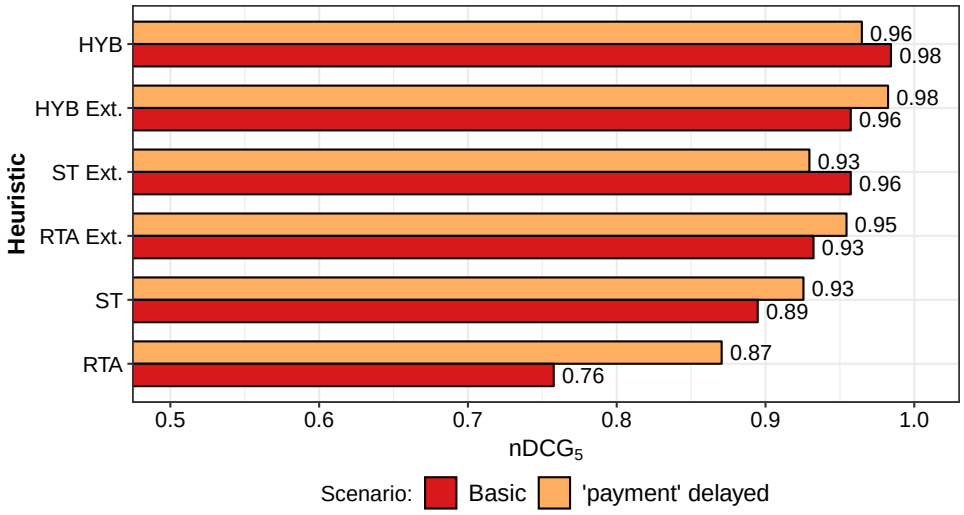

Figure 5.6: $nDCG_5$ scores for all variations of the three heuristics in two sub-scenarios: basic and delayed (in the canary variant). Hybrid approaches perform best.

Results. Figure 5.6 shows the nDCG scores of the three heuristics in their 6 variations for the *basic* and the *delayed* sub-scenarios. The *hybrid* variations outperform the other heuristics, though some other approaches achieve high scores as well. *RTA* produces good results for the *delayed* sub-scenario. However, it only captures the "relevance" of the delayed fragments and ignores the high relevance of the added functionality. This is simply because there are no performance issues associated with these changes. The addition of *uncertainty* for the *RTA Ext* variant helps to compensate this flaw and leads to stronger scores for both

sub-scenarios. Moreover, penalizing as a scoring factor turns out to have positive effects on the delayed sub-scenario. However, the standard *HYB* variant without penalties performs slightly better, though only by a whisker, e.g., by 0.005 on the combined score of both sub-scenarios for *HYB* and *HYB Ext.*

5.7.3 Scenario 2: Breaking Changes

The goal of the second scenario is to identify how the heuristics behave when dealing with more complex, cascading changes resulting in multiple version updates. This represents deployment scenarios and experiments dealing with multiple breaking API changes. Figure 5.7 depicts its topological difference in which *b* is the experiment's target service.

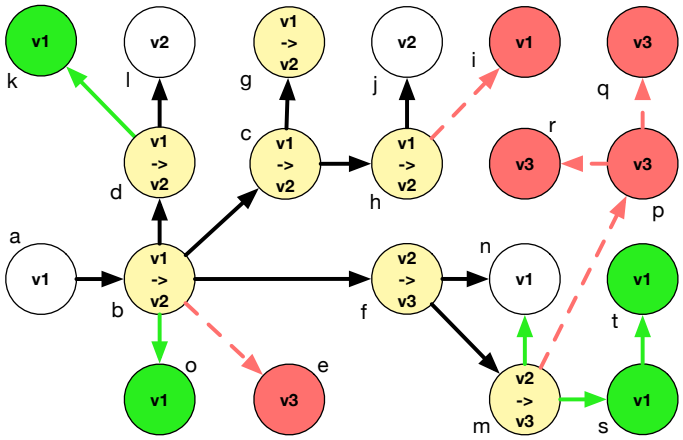

Figure 5.7: Topological differences of evaluation scenario 2.

Scenario. We split into multiple sub-scenarios involving simulated performance issues in the *canary* variant. In addition to the *basic* scenario, which contains multiple version updates and new services, we added two specific performance deviations: a delay at service *h* when calling service *j* (100ms), and a delay at service *s* (200*ms*) simulating a more complex request processing compared to the removed service pairs *p*, *q*, and *r*. As a fourth sub-scenario, we combined these two delays, making them active at the same time.

Relevance. For the *basic* sub-scenario, the version updates between b and c, b and f, f and m, and the added functionality for m calling s are rated as highly relevant. The delayed variants emphasize the changes introducing performance deviations.

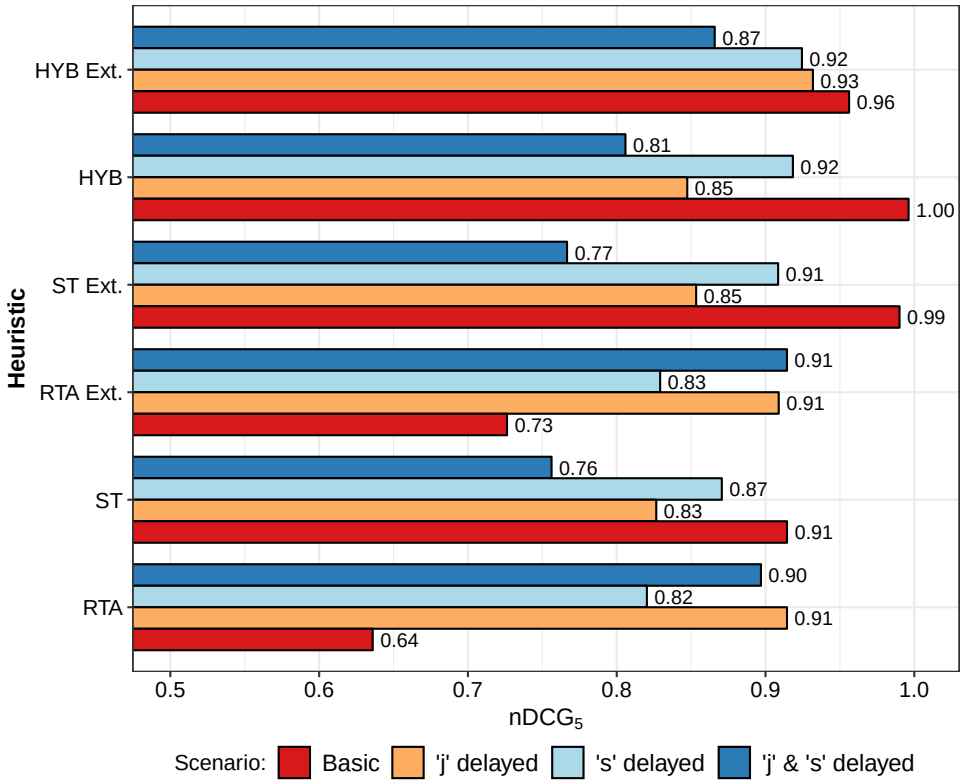

Figure 5.8: $nDCG_5$ scores for all variations of the three heuristics. Four sub-scenarios: basic, a delay involving service j (canary), a delay involving service s (canary), and a combination of both delays (canary).

Results. Similar to the running example, on average across all sub-scenarios, the *hybrid* heuristics perform best (see Figure 5.8). Some individual results on sub-scenarios provide valuable insights into the single heuristics' strengths and weaknesses. Keeping the *basic* results aside, *RTA* (in both variations) achieves an average nDCG score of 0.88, only topped by *HYB Ext*, which naturally inherited *RTA* functionality, with a score of 0.91. For the *basic* sub-scenario, the standard

HYB performs best, almost reporting the perfect ranking with a score of 0.996, immediately followed by *ST Ext* with *uncertainty* involved (as propagation and scoring factor). Remarkably, the standard version of *ST* achieves a score of 0.91, also due to the fact that changes rated with high relevance are particularly "up high in the tree" (e.g., between *b* and *f*, and *b* and *c*) in this scenario. This enables this simple heuristic to come close to the best rankings.

5.7.4 Discussion

Combining the nDCG scores across all evaluation scenarios yields the highest (average) score of 0.94 for *HYB Ext*, a heuristic involving both uncertainty and a penalty mechanism in the scoring function. Interestingly, when diving deeper and distinguishing between (1) all basic scenarios and (2) all scenarios involving introduced performance issues we observe *HYB Ext* being not ranked first for both (1) and (2). Despite being superior for performance cases (2) with an average score of 0.93 and a gap of 0.03 to the second-best heuristic (i.e., *RTA Ext*), it is ranked third for non-performance cases, lacking a score of 0.03 to its leading standard *HYB* counterpart without penalty mechanism. As the performance cases dominate – 4 versus 2 non-performance cases – *HYB Ext* clearly benefits from the evaluation setup. This result is an indication that it would make sense to let developers or release engineers using our proposed tooling toggle between multiple (selected) heuristics which provide insights onto the application's state from different angles.

5.8 Performance Evaluation

In the following we present results of the performance evaluation of the ranking heuristics for difference graphs of multiple sizes and with various characteristics.

5.8.1 Scope

Our performance evaluation focuses on the execution behavior of the heuristics. In contrast to the ranking quality evaluation we refrained from deploying case

study applications. Deploying applications with up to thousands of service endpoints as required for this evaluation setup, and generating traffic to allow the collection and analysis of its distributed traces with the ultimate goal to construct difference graphs, would not have been feasible within the budget and time constraints of this paper. Rather, we directly generate difference graphs that served as the input for our heuristics.

5.8.2 Setup

The setup describes how we created difference graphs for our evaluation and how and where we conducted the evaluation.

Difference Graphs. We were specifically interested in how the heuristics perform when the number of endpoints in the difference graphs increases. We created difference graphs (tooling available in our online appendix [Schermann et al., 2019]) of multiple sizes and with various characteristics. We randomly generated (cycle-free) graph structures of two types: *broad* and *deep*. For *broad* graphs, every endpoint calls on average 4.5 other endpoints following a Gaussian distribution with standard deviation (SD) of 1.5. We created *broad* graphs in step sizes from 2 to 11, where the step size represents the maximum path length from a node to the experiment's target node. Resulting graph sizes range from 13 to 113300 nodes. In case of *deep* graphs, every endpoint calls on average 2.25 other endpoints (again using a Gaussian distribution with SD 0.5). We created *deep* graphs in steps from 2 to 25, so that resulting graphs contain from 5 to 77300 nodes.

For every step size and every type we created four *change variants* of difference graphs. Change variants allow us to identify whether the heuristics differ in execution behavior for graphs with different "change frequencies", i.e., how many endpoints are added, removed, or updated. The first change variant is characterized by a small difference between baseline and canary versions, i.e., 2% of the generated endpoints in the resulting graph are marked as added, 2% as removed, and 2% as updated. The second change variant includes 5% added, 5% removed, and 10% updated endpoints. The third change variant includes 10% added, 15% removed, and 15% updated endpoints. Finally, the fourth

change variant includes 20% removed, 20% added, and 25% updated endpoints, resembling experiments for which the majority of services and endpoints in the application are changed. The higher number of updated endpoints in these scenarios reflects that adding or removing endpoints typically requires an update on the caller side (see Section 5.4).

For each of the change variants we created five *performance variants* of the graphs: containing no performance deviation at all, and containing performance deviations on 5%, 10%, 20%, and 30% of the endpoints. The intuition here is to reveal differences in the heuristics' execution behavior when dealing with spotted performance deviations. Both the endpoints causing performance deviations and the deviations' extents (between 30 and 200 ms) are randomly assigned.

Execution Setup and Environment. We generated 200 *broad* (10 steps * 4 change variants * 5 performance variants) and 480 *deep* ($24 * 4 * 5$) difference graphs. We then executed every heuristic (including standard and extended variants) on every generated graph 5 times. We measured overall execution times (in milliseconds, including time elapsed for reading difference graphs from the file system), heuristic execution times, and CPU and memory utilization.

We packaged the heuristics as standalone Node.js application (Node version 10.15.3). The evaluation was conducted on a IBM Cloud instance with 4 vCPUs and 8 GB of memory. All the generated difference graphs, the packaged Node.js application (including source code), and all the data analysis scripts are part of our extensive replication package Schermann et al. [2019].

5.8.3 Scalability of the Heuristics

To assess the scalability of our approach we are interested in how the heuristics perform under an increasing number of nodes in the difference graphs to be analyzed. Figure 5.9 visualizes how execution times evolve.

In general, the execution times of all variations of the heuristics are promising. Difference graphs consisting of less than 10,000 endpoints (e.g., 1,000 microservices with 10 endpoints each) can be analyzed within 5 seconds, graphs with 4,000 endpoints within 1 second. These results make our approach usable for

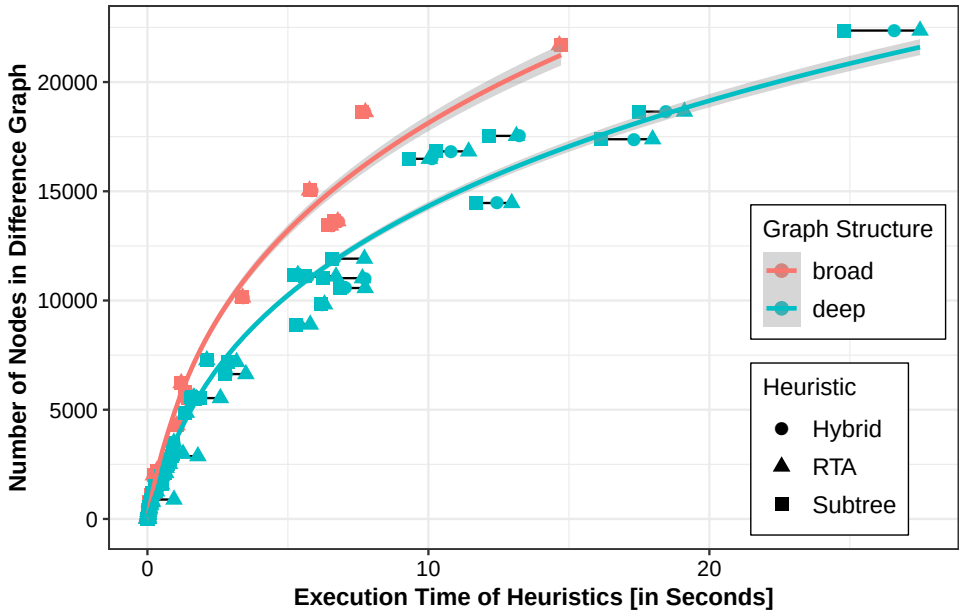

Figure 5.9: Heuristic execution times for increasing number of nodes in difference graphs with performance deviations on 30% of calls. Single data points represent mean execution time of the fastest variant of a heuristic (i.e., either standard or extended) for 5 repetitions. Trendlines based on polynomial regression.

our interactive tooling giving a quick overview about the main impacts of an experiment.

Looking at results for different graph structures in Figure 5.9, we identify that our approach is able to analyze more nodes in less time when the graph's structure is *broader* rather than *deeper*. The reason is that the deeper the structure – hence involving longer call traces – the more information needs to be propagated during the heuristics' analysis phases, which is time consuming. A similar effect surfaces when looking at the heuristics' execution times for a specific graph. In case of broad graphs, the best performing heuristics share very similar execution times, i.e., there is almost no gap between the heuristics. This is different for *deep* graphs, where we spot a slight gap as highlighted by black lines in Figure 5.9. For most of the graphs with more than 5,000 nodes *RTA* heuristics (cf. Section 5.5.4) are the slowest. We will investigate the heuristics' individual execution behavior in Section 5.8.4.

5.8.4 Individual Runtime Analysis

Breaking down the results onto the individual level for a specific difference graph gives us insights into (1) how stable the individual runs of the various heuristics are, and (2) how results change in the context of performance deviations.

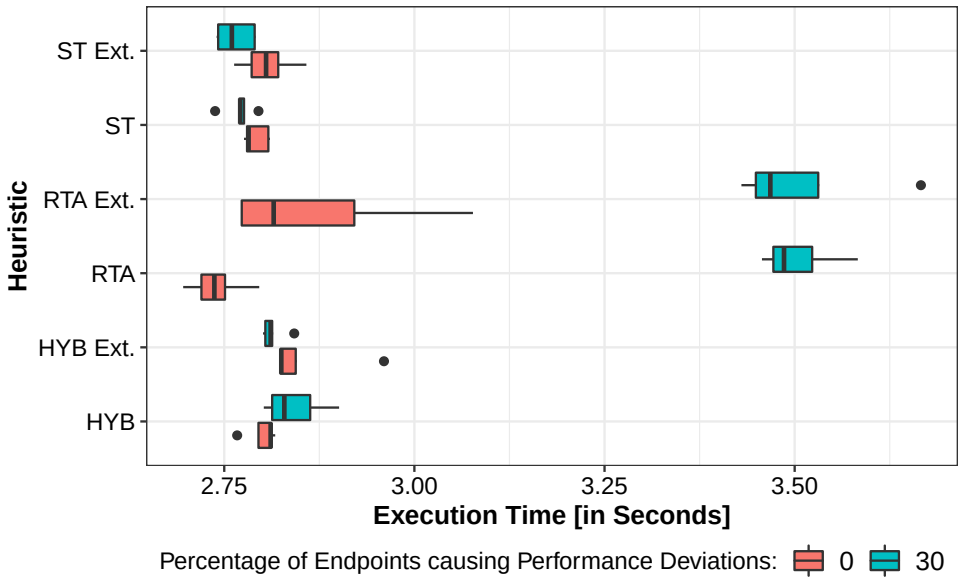

Figure 5.10: Box plots of heuristics' execution times for a *deep* difference graph with step size 19 (i.e., maximum path length from a node to the experiment's target service) and 6631 endpoints in total. Red shows execution times for the graph without any performance deviation, turquois with performance deviation on 30% of the endpoints.

In general, and not only bound to the specific step size and difference graph presented in Figure 5.10, the execution behavior of individual heuristics for the same input is very stable. Even for more complex graphs containing around 30,000 nodes – for which the analysis takes around 2 minutes – the standard deviation for individual heuristics lies in the order of 1 to 3 seconds.

However, we do observe differences in execution times for the same heuristic in presence of performance deviations. Figure 5.10 shows box plots of the runs for the same difference graph, once without and once including performance

deviations on 30% of the endpoints. While the analyses conducted with *ST* and *HYB* heuristics are only marginally slower or even faster on the graph with deviations, there is a performance reduction for *RTA* heuristics. Due to the cascading nature of these deviations (i.e., response time in our evaluation setup), a deviation caused on endpoint X causes a delay on any other endpoint calling endpoint X. Consequently, cases with 30% "malfunctioning" endpoints resemble worst case scenarios in which almost all endpoints are affected. This triggers *RTA* heuristics to analyze the surroundings of every affected node for determining the sources of the deviations and serves as an explanation for their slowdown. This is confirmed by our findings for the execution behavior on the same graph with performance deviations on 10% and 20% of the endpoints bridging the gap between the best (i.e., no deviation) and the worst results (i.e., 30% deviation). This behavior of the *RTA* heuristics is still surprising as *HYB* heuristics perform exactly the same analyses (i.e., same source code), the only difference is the combination with *ST* functionality. We suspect a Node.js-internal optimization that leads to this effect. However, the effect is quite small (i.e., less than a second in the majority of the cases) and has no influence on the entire approach.

5.8.5 Discussion

Overall, our heuristics show promising execution behavior for all the characteristics and sizes of difference graphs we analyzed. We did not spot any influence caused by the "change frequency" of a graph, i.e., the extent of changes between baseline and canary versions.

For the execution behavior of the whole approach including querying and clustering traces we can only rely on our case study applications created for the ranking quality evaluation. Querying and clustering 1,000 traces in total takes around 3 seconds (again based on 5 repetitions). Most time is consumed when querying ZipKin, while clustering traces only takes around 300 ms. Time spent on querying can be drastically reduced when only a small number of traces are sampled to construct the difference graphs. However, then, due to limited sample sizes, statistics computed during the clustering phase can not be relied on

and need to be replaced by queries to monitoring solutions such as Prometheus, which would cause additional delays.

Finally, memory and CPU utilization did not reveal abnormalities on graphs with different characteristics and sizes. Memory consumption is clearly driven by the size of the graphs, entire graphs are loaded into memory for analysis purposes. The approach characterizes as being more CPU than memory intensive. CPU utilization is at almost 100% during heuristic execution for graphs with multiple thousand nodes. The implementation itself provides space for improvement regarding parallelization, we expect better execution behavior when implemented in programming languages (with better suited mechanisms for parallel execution) different than our Node.js-based research prototype.

5.9 Limitations

One limitation of our approach is that the ranking quality evaluation was conducted on traces for self-generated scenarios. We mitigated this threat by covering two complex scenarios and combined them with sub-scenarios including simulated performance issues. A more thorough evaluation based on multiple real cases is desirable, and part of our future research. A further threat involves the relevance classification conducted by the authors of this paper. We classified all changes for all sub-scenarios on a scale from not relevant (0) to highly relevant (4). As the relevance is used as baseline for nDCG, these ratings have a direct effect on the resulting scores. Our replication package [Schermann et al., 2019] allows inspecting how results change when relevance ratings are adjusted. Another threat involves the parameter calibration for the heuristics, which has a strong influence on the results. We mitigated this threat by performing thorough calibration runs with different parameter settings across all covered scenarios.

One limitation regarding the heuristics is that *RTA* variations only account for changes that impact the response time negatively. We focus on the total response time, ignoring that individual changes can have both positive and negative effects. However, our heuristics can be extended fairly easily to cover this case as well.

A further limitation is that the performance evaluation only focused on the heuristics as an end-to-end evaluation for applications consisting of thousands of endpoints would have been not feasible. To provide a thorough assessment of the heuristics' execution behavior we generated difference graphs of multiple sizes and with various characteristics. Potential performance variations within the used cloud environments could have influenced our evaluation results [Leitner and Cito, 2016]. To mitigate this risk, we repeated every execution 5 times.

5.10 Conclusion

We proposed an approach that analyzes request traces captured from distributed tracing systems to identify changes of microservice-based applications in the context of continuous experiments. Using heuristics, we rank these identified changes according to their potential impact on the experiment and the application's health state, with the goal of supporting decisions on whether to continue or abort the experiment. While previous work on experiment health assessment considers the services under test in isolation, which could skew the assessment as certain effects are left out, we focus on the topological level. We characterized a set of reoccurring topological change types consisting of fundamental patterns and more complex composed variants. We proposed three heuristics that operate on top of these characterized changes taking the concept of *uncertainty* into account. Our evaluation conducted on two case study scenarios demonstrated that the rankings produced by the heuristics are promising and could be a valuable resource for experiment health assessments. Further, our evaluation on the heuristics' execution behavior showed that our approach scales well even for applications consisting of thousands of service endpoints.

Bibliography

Adams, B. and McIntosh, S. (2016). Modern Release Engineering in a Nutshell – Why Researchers should Care. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering (SANER), Future of Software Engineering (FOSE) Track*.

Ahmed, T. M., Bezemer, C.-P., Chen, T.-H., Hassan, A. E., and Shang, W. (2016). Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: An experience report. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 1–12, New York, NY, USA. ACM.

Arcuri, A. and Fraser, G. (2013). Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623.

Atkinson, R. and Flint, J. (2001). Accessing hidden and hard-to-reach populations: Snowball research strategies. *Social research update*, 33(1):1–4.

Bäck, T., Fogel, D. B., and Michalewicz, Z. (2000). *Evolutionary computation 1: Basic algorithms and operators*, volume 1. CRC press.

Bakshy, E., Eckles, D., and Bernstein, M. S. (2014). Designing and Deploying Online Field Experiments. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, pages 283–292, New York, NY, USA. ACM.

Bakshy, E. and Frachtenberg, E. (2015). Design and Analysis of Benchmarking Experiments for Distributed Internet Services. In *Proceedings of the 24th International Conference on World Wide Web (WWW)*, pages 108–118.

Barik, T., Johnson, B., and Murphy-Hill, E. (2015). I Heart Hacker News: Expanding Qualitative Research Findings by Analyzing Social News Websites. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 882–885, New York, NY, USA. ACM.

Bass, L., Weber, I., and Zhu, L. (2015). *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional.

Beck, K. (1999). Embracing change with extreme programming. *Computer*, 32(10):70–77.

Begel, A. and Zimmermann, T. (2014). Analyze This! 145 Questions for Data Scientists in Software Engineering. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 12–23, New York, NY, USA. ACM.

Beller, M., Gousios, G., and Zaidman, A. (2017). Oops, my tests broke the build: An explorative analysis of travis ci with github. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 356–367, Piscataway, NJ, USA. IEEE Press.

Bellomo, S., Ernst, N., Nord, R., and Kazman, R. (2014). Toward Design Decisions to Enable Deployability: Empirical Study of Three Projects Reaching for the Continuous Delivery Holy Grail. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 702–707.

Braa, K. and Vidgen, R. (1999). Interpretation, intervention, and reduction in the organizational laboratory: a framework for in-context information system research. *Accounting, Management and Information Technologies*, 9(1):25 – 47.

Brandtner, M., Giger, E., and Gall, H. (2014). Supporting continuous integration by mashing-up software quality information. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 184–193.

Brereton, P., Kitchenham, B., Budgen, D., and Li, Z. (2008). Using a protocol template for case study planning. In *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*, EASE'08, pages 41–48, Swindon, UK. BCS Learning & Development Ltd.

Bruneo, D., Fritz, T., Keidar-Barner, S., Leitner, P., Longo, F., Marquezan, C., Metzger, A., Pohl, K., Puliafito, A., Raz, D., Roth, A., Salant, E., Segall, I., Villari, M., Wolfsthal, Y., and Woods, C. (2014). CloudWave: where Adaptive Cloud Management Meets DevOps. In *Proceedings of the Fourth International Workshop on Management of Cloud Systems (MoCS 2014)*.

Capilla, R., Bosch, J., Kang, K.-C., et al. (2013). Systems and software variability management. *Concepts Tools and Experiences*.

Chen, L. (2015). Continuous Delivery: Huge Benefits, but Challenges Too. *Software, IEEE*, 32(2):50–54.

Chen, L. (2017). Continuous delivery: Overcoming adoption challenges. *Journal of Systems and Software*, 128:72 – 86.

Cito, J., Leitner, P., Fritz, T., and Gall, H. C. (2015a). The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 393–403, New York, NY, USA. ACM.

Cito, J., Leitner, P., Gall, H. C., Dadashi, A., Keller, A., and Roth, A. (2015b). Runtime Metric Meets Developer - Building Better Cloud Applications Using Feedback. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2015)*, New York, NY, USA. ACM.

Cito, J., Suljoti, D., Leitner, P., and Dustdar, S. (2014). Identifying root causes of web performance degradation using changepoint analysis. In Casteleyn, S., Rossi, G., and Winckler, M., editors, *Web Engineering*, pages 181–199, Cham. Springer International Publishing.

Claps, G. G., Svensson, R. B., and Aurum, A. (2015). On the Journey to Continuous Deployment: Technical and Social Challenges Along the Way. *Information and Software Technology*, 57(0):21 – 31.

Crook, T., Frasca, B., Kohavi, R., and Longbotham, R. (2009). Seven pitfalls to avoid when running controlled experiments on the web. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 1105–1114, New York, NY, USA. ACM.

Davidovic, S. and Beyer, B. (2018). Canary Analysis Service. *ACM Queue*, 16(1).

Deb, K. (2011). Multi-objective optimization using evolutionary algorithms: an introduction. *Multi-objective evolutionary optimisation for product design and manufacturing*, pages 1–24.

Debbiche, A., Dienér, M., and Berntsson Svensson, R. (2014). Challenges when adopting continuous integration: A case study. In *Product-Focused Software Process Improvement: 15th International Conference, PROFES 2014, Helsinki, Finland.*, pages 17–32. Springer International Publishing.

Etsy (2016). Code as Craft. `https://codeascraft.com/`. Accessed 2019-03-07.

European Commission (2014). Technology readiness levels (trl). `https://ec.europa.eu/research/participants/data/ref/h2020/wp/2014_2015/annexes/h2020-wp1415-annex-g-trl_en.pdf`. Accessed 2019-03-10.

Fabijan, A., Dmitriev, P., Holmstrom Olsson, H., and Bosch, J. (2018). The online controlled experiment lifecycle. *IEEE Software*, pages 1–1.

Fabijan, A., Dmitriev, P., Olsson, H. H., and Bosch, J. (2017). The evolution of continuous experimentation in software product development: From data to

a data-driven organization at scale. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 770–780.

Facebook (2016). Engineering Blog. `https://code.facebook.com/posts/`. Accessed 2019-03-07.

Fagerholm, F., Guinea, A. S., Mäenpää, H., and Münch, J. (2017). The right model for continuous experimentation. *Journal of Systems and Software*, 123:292 – 305.

Feitelson, D. G., Frachtenberg, E., and Beck, K. L. (2013). Development and Deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17.

Fitzgerald, B. and Stol, K.-J. (2014). Continuous software engineering and beyond: Trends and challenges. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, RCoSE 2014, pages 1–9, New York, NY, USA. ACM.

Fonseca, C. M. and Fleming, P. J. (1993). Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 416–423, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. (2007). X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association.

Foo, K. C., Jiang, Z. M. J., Adams, B., Hassan, A. E., Zou, Y., and Flora, P. (2015). An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 159–168, Piscataway, NJ, USA. IEEE Press.

Fowler, M. (2013). Continuous Delivery. `http://martinfowler.com/bliki/ContinuousDelivery.html`. Accessed 2019-03-07.

Frey, S., Fittkau, F., and Hasselbring, W. (2013). Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 512–521, Piscataway, NJ, USA. IEEE Press.

Froemmgen, A., Stohr, D., Koldehofe, B., and Rizk, A. (2018). Don't Repeat Yourself: Seamless Execution and Analysis of Extensive Network Experiments. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT'18)*.

Fu, W., Menzies, T., and Shen, X. (2016). Tuning for software analytics: Is it really necessary? *Information and Software Technology*, 76:135 – 146.

Galster, M., Weyns, D., Tofan, D., Michalik, B., and Avgeriou, P. (2014). Variability in software systems - a systematic literature review. *IEEE Transactions on Software Engineering*, 40(3):282–306.

Garousi, V., Felderer, M., and Mäntylä, M. V. (2016). The need for multivocal literature reviews in software engineering: Complementing systematic literature reviews with grey literature. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, pages 26:1–26:6, New York, NY, USA. ACM.

Goldberg, D. E. and Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. volume 1 of *Foundations of Genetic Algorithms*, pages 69 – 93. Elsevier.

Gomez-Uribe, C. A. and Hunt, N. (2016). The netflix recommender system: Algorithms, business value, and innovation. *ACM Transactions on Management Information Systems (TMIS)*, 6(4):13.

Google (2016). Google Developers Blog. `https://developers.googleblog.com/`. Accessed 2019-03-07.

Hallsteinsen, S., Hinchey, M., Park, S., and Schmid, K. (2008). Dynamic Software Product Lines. *Computer*, 41(4):93–95.

Harman, M. (2011). Software engineering meets evolutionary computation. *Computer*, 44(10):31–39.

Hilton, M., Nelson, N., Tunnell, T., Marinov, D., and Dig, D. (2017). Trade-offs in continuous integration: Assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 197–207, New York, NY, USA. ACM.

Hilton, M., Tunnell, T., Huang, K., Marinov, D., and Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 426–437, New York, NY, USA. ACM.

Hodgson, P. (2016). Feature Toggles. `http://martinfowler.com/articles/feature-toggles.html`. Accessed 2019-03-07.

Hohpe, G., Ozkaya, I., Zdun, U., and Zimmermann, O. (2016). The software architect's role in the digital age. *IEEE Software*, 33(6):30–39.

Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional.

Humble, J., Molesky, J., and O'Reilly, B. (2015). *Lean Enterprise: How High Performance Organizations Innovate at Scale*. O'Reilly Media, Inc., 1st edition.

Hummer, W., Rosenberg, F., Oliveira, F., and Eilam, T. (2013). Testing idempotence for infrastructure as code. In Eyers, D. and Schwan, K., editors, *Middleware 2013: ACM/IFIP/USENIX 14th International Middleware Conference, Beijing, China, December 9-13, 2013, Proceedings*, pages 368–388, Berlin, Heidelberg. Springer Berlin Heidelberg.

Istio (2019). `https://istio.io/`. Accessed 2019-03-07.

Itkonen, J., Udd, R., Lassenius, C., and Lehtonen, T. (2016). Perceived benefits of adopting continuous delivery practices. In *Proceedings of the 10th ACM/IEEE*

International Symposium on Empirical Software Engineering and Measurement, ESEM '16, pages 42:1–42:6, New York, NY, USA. ACM.

Jaeger (2019). Jaeger: open source, end-to-end distributed tracing. `https://www.jaegertracing.io/`. Accessed 2019-03-07.

Järvelin, K. and Kekäläinen, J. (2002). Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446.

Kevic, K., Murphy, B., Williams, L., and Beckmann, J. (2017). Characterizing experimentation in continuous deployment: A case study on bing. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ICSE-SEIP '17, pages 123–132, Piscataway, NJ, USA. IEEE Press.

Kiczales, G. (1996). Aspect-oriented Programming. *ACM Computing Surveys*, 28(4).

Kim, M., Zimmermann, T., DeLine, R., and Begel, A. (2016). The Emerging Role of Data Scientists on Software Development Teams. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 96–107, New York, NY, USA. ACM.

Kohavi, R., Deng, A., Frasca, B., Walker, T., Xu, Y., and Pohlmann, N. (2013). Online Controlled Experiments at Large Scale. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 1168–1176, New York, NY, USA. ACM.

Kohavi, R., Deng, A., Longbotham, R., and Xu, Y. (2014). Seven rules of thumb for web site experimenters. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14. ACM.

Kohavi, R., Henne, R. M., and Sommerfield, D. (2007). Practical guide to controlled experiments on the web: Listen to your customers not to the hippo. In *Proceedings of the 13th ACM SIGKDD International Conference on*

Knowledge Discovery and Data Mining, KDD '07, pages 959–967, New York, NY, USA. ACM.

Kohavi, R., Longbotham, R., Sommerfield, D., and Henne, R. M. (2009). Controlled experiments on the web: survey and practical guide. *Data Mining and Knowledge Discovery*, 18(1):140–181.

Kruchten, P., Nord, R. L., and Ozkaya, I. (2012). Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6):18–21.

Kubernetes (2019). `https://kubernetes.io/`. Accessed 2019-06-25.

Leitner, P. and Cito, J. (2016). Patterns in the chaos - a study of performance variation and predictability in public iaas clouds. *ACM Trans. Internet Technol.*, 16(3):15:1–15:23.

Leppanen, M., Makinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mantyla, M., and Mannisto, T. (2015). The Highways and Country Roads to Continuous Deployment. *IEEE Software*, 32(2):64–72.

Lethbridge, T. C., Sim, S. E., and Singer, J. (2005). Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering*, 10(3):311–341.

Lewis, J. and Fowler, M. (2014). Microservices. `http://martinfowler.com/articles/microservices.html`. Accessed 2019-03-07.

Lin, B., Zagalsky, A., Storey, M., and Serebrenik, A. (2016). Why Developers Are Slacking Off: Understanding How Software Teams Use Slack. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW) Companion*, pages 333–336, New York, NY, USA. ACM.

Lindgren, E. and Münch, J. (2016). Raising the Odds of Success: the Current State of Experimentation in Product Development. *Information and Software Technology*, 77:80 – 91.

Linkerd (2019). `https://linkerd.io/`. Accessed 2019-03-07.

Lwakatare, L. E., Kuvaja, P., and Oivo, M. (2015). Dimensions of devops. In *International Conference on Agile Software Development*, pages 212–217. Springer.

Lwakatare, L. E., Kuvaja, P., and Oivo, M. (2016). Relationship of devops to agile, lean and continuous deployment: A multivocal literature review study. In *Product-Focused Software Process Improvement: 17th International Conference, PROFES 2016, Trondheim, Norway.*, pages 399–415. Springer.

Mazlami, G., Cito, J., and Leitner, P. (2017). Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531.

McSherry, F. and Najork, M. (2008). Computing information retrieval performance measures efficiently in the presence of tied scores. In Macdonald, C., Ounis, I., Plachouras, V., Ruthven, I., and White, R. W., editors, *Advances in Information Retrieval*, pages 414–421, Berlin, Heidelberg. Springer Berlin Heidelberg.

Menascé, D. A. (2002). Load testing of web sites. *IEEE Internet Computing*, 6(4):70–74.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and How to Develop Domain-specific Languages. *ACM Computing Surveys*, 37(4):316–344.

Moskowitz, A. (2003). Eat Your Own Dog Food. *j-LOGIN*, 28(5).

Mostafa, S., Wang, X., and Xie, T. (2017). Perfranker: Prioritization of performance regression tests for collection-intensive software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 23–34, New York, NY, USA. ACM.

Neely, S. and Stolt, S. (2013). Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy). In *Agile Conference (AGILE), 2013*, pages 121–128.

Netflix (2016). The Netflix Tech Blog. `http://techblog.netflix.com/`. Accessed 2019-03-07.

Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc., 1st edition.

Olsson, H., Alahyari, H., and Bosch, J. (2012). Climbing the "Stairway to Heaven" – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In *Proceedings of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 392–399.

Open Tracing (2019). `https://opentracing.io/`. Accessed 2019-03-07.

Parnin, C., Helms, E., Atlee, C., Boughton, H., Ghattas, M., Glover, A., Holman, J., Micco, J., Murphy, B., Savor, T., Stumm, M., Whitaker, S., and Williams, L. (2017). The top 10 adages in continuous deployment. *IEEE Software*, 34(3):86–95.

Prometheus (2019). `https://prometheus.io/`. Accessed 2019-03-07.

Provost, F. and Fawcett, T. (2013). Data Science and its Relationship to Big Data and Data-Driven Decision Making. *Big Data*, 1(1):51–59.

Puppet Labs (2016). State of DevOps Report. `https://puppetlabs.com/2016-devops-report`. Accessed 2019-03-07.

Rahman, A., Helms, E., Williams, L., and Parnin, C. (2015). Synthesizing Continuous Deployment Practices Used in Software Development. In *Agile Conference (AGILE), 2015*, pages 1–10.

Rahman, M. T., Querel, L.-P., Rigby, P. C., and Adams, B. (2016). Feature Toggles: Practitioner Practices and a Case Study. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 201–211, New York, NY, USA. ACM.

Rausch, T., Hummer, W., Leitner, P., and Schulte, S. (2017). An empirical analysis of build failures in the continuous integration workflows of java-based

open-source software. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 345–355, Piscataway, NJ, USA. IEEE Press.

Rodríguez, P., Haghighatkhah, A., Lwakatare, L. E., Teppola, S., Suomalainen, T., Eskeli, J., Karvonen, T., Kuvaja, P., Verner, J. M., and Oivo, M. (2016). Continuous Deployment of Software Intensive Products and Services: A Systematic Mapping Study. *Journal of Systems and Software*.

Romano, D., Raemaekers, S., and Pinzger, M. (2014). Refactoring fat interfaces using a genetic algorithm. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 351–360.

Rubin, J. and Rinard, M. (2016). The Challenges of Staying Together While Moving Fast: An Exploratory Study. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 982–993, New York, NY, USA. ACM.

Runeson, P., Höst, M., Rainer, A., and Regnell, B. (2012). *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Publishing, 1st edition.

Sambasivan, R. R., Zheng, A. X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., and Ganger, G. R. (2011). Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 43–56, Berkeley, CA, USA. USENIX Association.

Sarro, F., Petrozziello, A., and Harman, M. (2016). Multi-objective software effort estimation. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 619–630, New York, NY, USA. ACM.

Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., and Stumm, M. (2016). Continuous Deployment at Facebook and OANDA. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pages 21–30, New York, NY, USA. ACM.

Schermann, G., Cito, J., and Leitner, P. (2015). All the services large and micro: Revisiting industrial practice in services computing. In Norta, A., Gaaloul, W., Gangadharan, G. R., and Dam, H. K., editors, *Service-Oriented Computing – ICSOC 2015 Workshops*, pages 36–47, Berlin, Heidelberg. Springer Berlin Heidelberg.

Schermann, G., Cito, J., and Leitner, P. (2018a). Continuous experimentation: Challenges, implementation techniques, and current research. *IEEE Software*, 35(2):26–31.

Schermann, G., Cito, J., Leitner, P., and Gall, H. C. (2016a). Towards Quality Gates in Continuous Delivery and Deployment. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–4. IEEE.

Schermann, G., Cito, J., Leitner, P., Zdun, U., and Gall, H. C. (2018b). We're doing it live: A multi-method empirical study on continuous experimentation. *Information and Software Technology*, 99:41 – 57.

Schermann, G. and Leitner, P. (2018). Paper Online Appendix. `https://github.com/sealuzh/icsme18-fenrir`. Accessed 2019-03-07.

Schermann, G., Oliveira, F., Wittern, E., and Leitner, P. (2019). Online Appendix. `https://github.com/anon-researcher/experimentation`. Accessed 2019-03-07.

Schermann, G., Schöni, D., Leitner, P., and Gall, H. C. (2016b). Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, pages 12:1–12:14, New York, NY, USA. ACM.

Schoen, F. (1991). Stochastic techniques for global optimization: A survey of recent advances. *Journal of Global Optimization*, 1(3):207–228.

Shahin, M., Babar, M. A., and Zhu, L. (2016). The intersection of continuous deployment and architecting process: Practitioners' perspectives. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software*

Engineering and Measurement, ESEM '16, pages 44:1–44:10, New York, NY, USA. ACM.

Shahin, M., Babar, M. A., and Zhu, L. (2017a). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943.

Shahin, M., Zahedi, M., Babar, M. A., and Zhu, L. (2017b). Adopting continuous delivery and deployment: Impacts on team structures, collaboration and responsibilities. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 384–393. ACM.

Shull, F., Singer, J., and Sjøberg, D. I. (2007). *Guide to Advanced Empirical Software Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. (2010). Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc.

Smith, E., Loftin, R., Murphy-Hill, E., Bird, C., and Zimmermann, T. (2013). Improving Developer Participation Rates in Surveys. In *Proceedings of the 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 89–92.

Spencer, D. (2009). *Card Sorting: Designing Usable Categories*. Rosenfeld Media.

Ståhl, D. and Bosch, J. (2014). Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48 – 59.

Tamburrelli, G. and Margara, A. (2014). Towards Automated A/B Testing. In *Proceedings of the 6th International Symposium on Search-Based Software Engineering (SSBSE)*, volume 8636 of *Lecture Notes in Computer Science*, pages 184–198. Springer.

Tang, C., Kooburat, T., Venkatachalam, P., Chander, A., Wen, Z., Narayanan, A., Dowell, P., and Karl, R. (2015). Holistic Configuration Management

at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, pages 328–343, New York, NY, USA. ACM.

Tang, D., Agarwal, A., O'Brien, D., and Meyer, M. (2010). Overlapping experiment infrastructure: More, better, faster experimentation. In *Proceedings 16th Conference on Knowledge Discovery and Data Mining*, pages 17–26, Washington, DC.

Tantithamthavorn, C., McIntosh, S., Hassan, A. E., and Matsumoto, K. (2018). The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, pages 1–1.

Tarvo, A., Sweeney, P. F., Mitchell, N., Rajan, V., Arnold, M., and Baldini, I. (2015). CanaryAdvisor: A Statistical-based Tool for Canary Testing (Demo). In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pages 418–422, New York, NY, USA. ACM.

ThoughtWorks and Forrester Consulting (2013). Continuous Delivery: A Maturity Assessment Model. `https://info.thoughtworks.com/Continuous-Delivery-Maturity-Model.html`. Accessed 2019-03-07.

Twitter (2016). The Twitter Engineering Blog. `https://blog.twitter.com/engineering`. Accessed 2019-03-07.

van Hoorn, A., Waller, J., and Hasselbring, W. (2012). Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE'12)*, pages 247–248, New York, NY, USA. ACM.

Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., and Filkov, V. (2015). Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 805–816, New York, NY, USA. ACM.

Vassallo, C., Schermann, G., Zampetti, F., Romano, D., Leitner, P., Zaidman, A., Penta, M. D., and Panichella, S. (2017). A tale of ci build failures: An open source and a financial organization perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 183–193.

Veeraraghavan, K., Meza, J., Chou, D., Kim, W., Margulis, S., Michelson, S., Nishtala, R., Obenshain, D., Perelman, D., and Song, Y. J. (2016). Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 635–650, Berkeley, CA, USA. USENIX Association.

Wall, M. B. (1996). *A genetic algorithm for resource-constrained scheduling*. PhD thesis, Massachusetts Institute of Technology.

Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. (2009). Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA. IEEE Computer Society.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., and Wesslén, A. (2000). *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers.

Xanthakis, S., Ellis, C., Skourlas, C., Le Gall, A., Katsikas, S., and Karapoulios, K. (1992). Application of genetic algorithms to software testing. In *Proceedings of the 5th International Conference on Software Engineering and Applications*, pages 625–636.

Xu, Y., Chen, N., Fernandez, A., Sinno, O., and Bhasin, A. (2015). From infrastructure to culture: A/b testing challenges in large scale social networks. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '15, pages 2227–2236, New York, NY, USA. ACM.

Zipkin (2019). `https://zipkin.io/`. Accessed 2019-03-07.

Curriculum Vitae

Personal Information

Name	Gerald Schermann
Nationality	Austria
Date of Birth	February 26, 1989

Education

2014 – 2019	Doctoral Program in Informatics Department of Informatics University of Zurich, Switzerland
2012 – 2014	Master of Science in Computer Science Vienna University of Technology, Austria
2009 – 2012	Bachelor of Science in Computer Science Vienna University of Technology, Austria